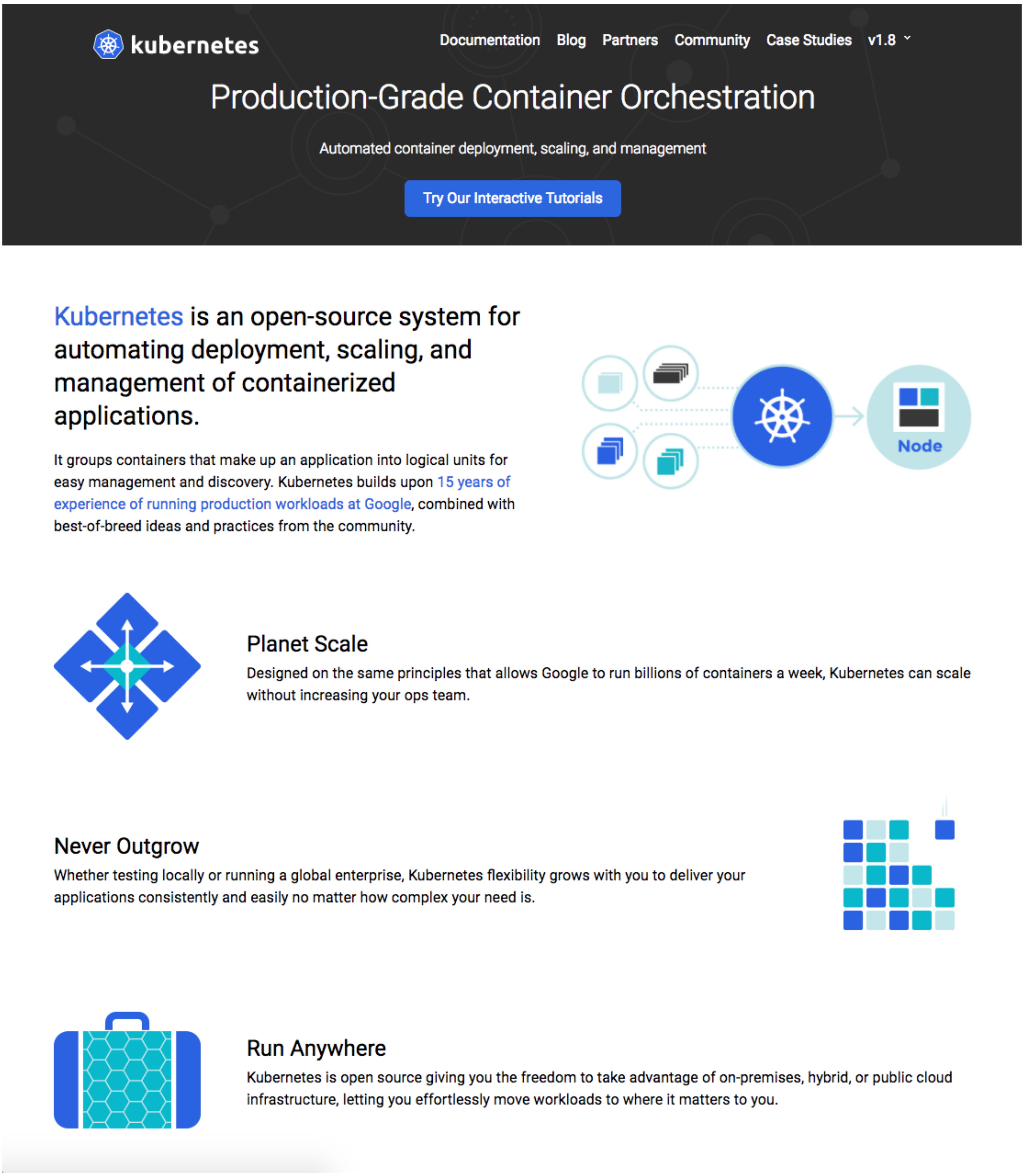


Production-Grade Container Orchestration

Automated container deployment, scaling, and management

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Planet Scale

Designed on the same principles that allows Google to run billions of containers a week, Kubernetes can scale without increasing your ops team.

Never Outgrow

Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is.

Run Anywhere

Kubernetes is open source giving you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.

Kubernetes Documentation

Kubernetes documentation can help you set up Kubernetes, learn about the system, or get your applications and workloads running on Kubernetes. To learn the basics of what Kubernetes is and how it works, read “What is Kubernetes”.

Interactive Tutorial

The Kubernetes Basics interactive tutorial lets you try out Kubernetes right out of your web browser, using a virtual terminal. Learn about the Kubernetes system and deploy, expose, scale, and upgrade a containerized application in just a few minutes.

Installing/Setting Up Kubernetes

Picking the Right Solution can help you get a Kubernetes cluster up and running, either for local development, or on your cloud provider of choice.

Concepts, Tasks, and Tutorials

The Kubernetes documentation contains a number of resources to help you understand and work with Kubernetes.

- Concepts provide a deep understanding of how Kubernetes works.
- Tasks contain step-by-step instructions for common Kubernetes tasks.
- Tutorials contain detailed walkthroughs of the Kubernetes workflow.

API and Command References

The Reference documentation provides complete information on the Kubernetes APIs and the `kubectl` command-line interface.

Tools

The Tools page contains a list of native and third-party tools for Kubernetes.

Troubleshooting

The Troubleshooting page outlines some resources for troubleshooting and finding help.

Troubleshooting

Sometimes things go wrong. This guide is aimed at making them right. It has two sections:

- Troubleshooting your application - Useful for users who are deploying code into Kubernetes and wondering why it is not working.
- Troubleshooting your cluster - Useful for cluster administrators and people whose Kubernetes cluster is unhappy.

You should also check the known issues for the release you're using.

Getting help

If your problem isn't answered by any of the guides above, there are variety of ways for you to get help from the Kubernetes team.

Questions

The documentation on this site has been structured to provide answers to a wide range of questions. Concepts explain the Kubernetes architecture and how each component works, while Setup provides practical instructions for getting started. Tasks show how to accomplish commonly used tasks, and Tutorials are more comprehensive walkthroughs of real-world, industry-specific, or end-to-end development scenarios. The Reference section provides detailed documentation on the Kubernetes API and command-line interfaces (CLIs), such as `kubectl`.

You may also find the Stack Overflow topics relevant:

- Kubernetes
- Google Container Engine - GKE

Help! My question isn't covered! I need help now!

Stack Overflow

Someone else from the community may have already asked a similar question or may be able to help with your problem. The Kubernetes team will also monitor posts tagged Kubernetes. If there aren't any existing questions that help, please ask a new one!

Slack

The Kubernetes team hangs out on Slack in the `#kubernetes-users` channel. You can participate in discussion with the Kubernetes team here. Slack requires registration, but the Kubernetes team is open invitation to anyone to register here. Feel free to come and ask any and all questions.

Once registered, browse the growing list of channels for various subjects of interest. For example, people new to Kubernetes may also want to join the `#kubernetes-novice` channel. As another example, developers should join the `#kubernetes-dev` channel.

There are also many country specific/local language channels. Feel free to join these channels for localized support and info:

- France: `#fr-users`, `#fr-events`
- Germany: `#de-users`, `#de-events`
- Japan: `#jp-users`, `#jp-events`

Mailing List

The Kubernetes / Google Container Engine mailing list is kubernetes-users@googlegroups.com

Bugs and Feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the Github issue tracking system.

Before you file an issue, please search existing issues to see if your issue is already covered.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: `kubectl version`

- Cloud provider, OS distro, network configuration, and Docker version
- Steps to reproduce the problem

Creating a Documentation Pull Request

To contribute to the Kubernetes documentation, create a pull request against the kubernetes/kubernetes.github.io repository. This page shows how to create a pull request.

Before you begin

1. Create a GitHub account.

2. Sign the Linux Foundation Contributor License Agreement.

Documentation will be published under the CC BY SA 4.0 license.

Creating a fork of the Kubernetes documentation repository

1. Go to the kubernetes/kubernetes.github.io repository.

2. In the upper-right corner, click **Fork**. This creates a copy of the Kubernetes documentation repository in your GitHub account. The copy is called a *fork*.

Making your changes

1. In your GitHub account, in your fork of the Kubernetes docs, create a new branch to use for your contribution.

2. In your new branch, make your changes and commit them. If you want to write a new topic, choose the page type that is the best fit for your content.

Submitting a pull request to the master branch (Current Release)

If you want your change to be published in the released version Kubernetes docs, create a pull request against the master branch of the Kubernetes documentation repository.

1. In your GitHub account, in your new branch, create a pull request against the master branch of the kubernetes/kubernetes.github.io repository. This opens a page that shows the status of your pull request.
2. Click **Show all checks**. Wait for the **deploy/netlify** check to complete. To the right of **deploy/netlify**, click **Details**. This opens a staging site where you can verify that your changes have rendered correctly.
3. During the next few days, check your pull request for reviewer comments. If needed, revise your pull request by committing changes to your new branch in your fork.

Submitting a pull request to the <vnext> branch (Upcoming Release)

If your documentation change should not be released until the next release of the Kubernetes product, create a pull request against the <vnext> branch of the Kubernetes documentation repository. The <vnext> branch has the form `release-<version-number>`, for example release-1.5.

1. In your GitHub account, in your new branch, create a pull request against the <vnext> branch of the kubernetes/kubernetes.github.io repository. This opens a page that shows the status of your pull request.
2. Click **Show all checks**. Wait for the **deploy/netlify** check to complete. To the right of **deploy/netlify**, click **Details**. This opens a staging site where you can verify that your changes have rendered correctly.

3. During the next few days, check your pull request for reviewer comments. If needed, revise your pull request by committing changes to your new branch in your fork.

The staging site for the upcoming Kubernetes release is here: http://kubernetes-io-vnext-staging.netlify.com/. The staging site reflects the current state of what's been merged in the release branch, or in other words, what the docs will look like for the next upcoming release. It's automatically updated as new PRs get merged.

What's next

- Learn about writing a new topic.
- Learn about using page templates.
- Learn about staging your changes.

Writing a New Topic

This page shows how to create a new topic for the Kubernetes docs.

Before you begin

Create a fork of the Kubernetes documentation repository as described in Creating a Documentation Pull Request.

Choosing a page type

As you prepare to write a new topic, think about which of these page types is the best fit for your content:

Task	A task page shows how to do a single thing. The idea is to give readers a sequence of steps that they can actually do as they read the page. A task page can be short or long, provided it stays focused on one area. In a task page, it is OK to blend brief explanations with the steps to be performed, but if you need to provide a lengthy explanation, you should do that in a concept topic. Related task and concept topics should link to each other. For an example of a short task page, see Configure a Pod to Use a Volume for Storage. For an example of a longer task page, see Configure Liveness and Readiness Probes
Tutorial	A tutorial page shows how to accomplish a goal that ties together several Kubernetes features. A tutorial might provide several sequences of steps that readers can actually do as they read the page. Or it might provide explanations of related pieces of code. For example, a tutorial could provide a walkthrough of a code sample. A tutorial can include brief explanations of the Kubernetes features that are being tied togeter, but should link to related concept topics for deep explanations of individual features.
Concept	A concept page explains some aspect of Kubernetes. For example, a concept page might describe the Kubernetes

Deployment object and explain the role it plays as an application is deployed, scaled, and updated. Typically, concept pages don't include sequences of steps, but instead provide links to tasks or tutorials. For an example of a concept topic, see Nodes.

Each page type has a template that you can use as you write your topic. Using templates helps ensure consistency among topics of a given type.

Choosing a title and filename

Choose a title that has the keywords you want search engines to find. Create a filename that uses the words in your title separated by hyphens. For example, the topic with title Using an HTTP Proxy to Access the Kubernetes API has filename `http-proxy-access-api.md`. You don't need to put "kubernetes" in the filename, because "kubernetes" is already in the URL for the topic, for example:

```
http://kubernetes.io/docs/tasks/access-kubernetes-api/http-proxy-access-api/
```

Adding the topic title to the front matter

In your topic, put a `title` field in the front matter. The front matter is the YAML block that is between the triple-dashed lines at the top of the page. Here's an example:

```
---
title: Using an HTTP Proxy to Access the Kubernetes API
---
```

Choosing a directory

Depending on your page type, put your new file in a subdirectory of one of these:

- /docs/tasks/
- /docs/tutorials/
- /docs/concepts/

You can put your file in an existing subdirectory, or you can create a new subdirectory.

Creating an entry in the table of contents

Depending page type, create an entry in one of these files:

- /_data/tasks.yaml
- /_data/tutorials.yaml
- /_data/concepts.yaml

Here's an example of an entry in /_data/tasks.yaml:

```
- docs/tasks/configure-pod-container/configure-volume-storage.md
```

Including code from another file

To include a code file in your topic, place the code file in the Kubernetes documentation repository, preferably in the same directory as your topic file. In your topic file, use the `include` tag:

```
{% include code.html language="<LEXERVALUE>" file="<RELATIVEPATH>" ghlink="/<PATHF
```

where:

- `<LEXERVALUE>` is the language in which the file was written. This must be a value supported by Rouge.
- `<RELATIVEPATH>` is the path to the file you're including, relative to the current file, for example, `gce-volume.yaml`.
- `<PATHFROMROOT>` is the path to the file relative to root, for example, `docs/tutorials/stateful-application/gce-volume.yaml`.

Here's an example of using the `include` tag:

```
{% include code.html language="yaml" file="gce-volume.yaml" ghlink="/docs/tutorial
```

Showing how to create an API object from a configuration file

If you need to show the reader how to create an API object based on a configuration file, place the configuration file in the Kubernetes documentation repository, preferably in the same directory as your topic file.

In your topic, show this command:

```
kubectl create -f https://k8s.io/<PATHFROMROOT>
```

where `<PATHFROMROOT>` is the path to the configuration file relative to root, for example, `docs/tutorials/stateful-application/gce-volume.yaml`.

Here's an example of a command that creates an API object from a configuration file:

```
kubectl create -f https://k8s.io/docs/tutorials/stateful-application/gce-volume.ya
```

For an example of a topic that uses this technique, see Running a Single-Instance Stateful Application.

Adding images to a topic

Put image files in the `/images` directory. The preferred image format is SVG.

What's next

- Learn about using page templates.
- Learn about staging your changes.
- Learn about creating a pull request.

Staging Your Documentation Changes

This page shows how to stage content that you want to contribute to the Kubernetes documentation.

Before you begin

Create a fork of the Kubernetes documentation repository as described in Creating a Documentation Pull Request.

Staging a pull request

When you create a pull request, either against the master or <vnext> branch, your changes are staged in a custom subdomain on Netlify so that you can see your changes in rendered form before the pull request is merged.

1. In your GitHub account, in your new branch, submit a pull request to the kubernetes/kubernetes.github.io repository. This opens a page that shows the status of your pull request.
2. Scroll down to the list of automated checks. Click **Show all checks**. Wait for the **deploy/netlify** check to complete. To the right of **deploy/netlify**, click **Details**. This opens a staging site where you can see your changes.

Staging locally using Docker

You can use the k8sdocs Docker image to run a local staging server. If you're interested, you can view the Dockerfile for this image.

1. Install Docker if you don't already have it.
2. Clone your fork to your local development machine.
3. In the root of your cloned repository, enter this command to start a local web server:

   ```
   make stage
   ```

   This will run the following command:

   ```
   docker run -ti --rm -v "$PWD":/k8sdocs -p 4000:4000 gcr.io/google-samples/k8sd
   ```

4. View your staged content at `http://localhost:4000`.

Staging locally without Docker

1. Install Ruby 2.2 or later.
2. Install RubyGems.
3. Verify that Ruby and RubyGems are installed:

   ```
   gem --version
   ```

4. Install the GitHub Pages package, which includes Jekyll:

   ```
   gem install github-pages
   ```

5. Clone your fork to your local development machine.
6. In the root of your cloned repository, enter this command to start a local web server:

```
jekyll serve
```

7. View your staged content at `http://localhost:4000`.

> **Note:** "If you do not want Jekyll to interfere with your other globally installed gems, you can use `bundler`:
>
> `gem install bundler`
> `bundle install`
> `bundler exec jekyll serve`
>
> Regardless of whether you use `bundler` or not, your copy of the site will then be viewable at: http://localhost:4000

What's next

- Learn about writing a new topic.
- Learn about using page templates.
- Learn about creating a pull request.

Using Page Templates

These page templates are available for writers who would like to contribute new topics to the Kubernetes docs:

- Task
- Tutorial
- Concept

The page templates are in the _includes/templates directory of the kubernetes.github.io repository.

Task template

A task page shows how to do a single thing, typically by giving a short sequence of steps. Task pages have minimal explanation, but often provide links to conceptual topics that provide related background and knowledge.

To write a new task page, create a Markdown file in a subdirectory of the /docs/tasks directory. In your Markdown file, provide values for these variables:

- overview - required
- prerequisites - required
- steps - required
- discussion - optional
- whatsnext - optional

Then include templates/task.md like this:

```
...
{% include templates/task.md %}
```

In the `steps` section, use `##` to start with a level-two heading. For subheadings, use `###` and `####` as needed. Similarly, if you choose to have a `discussion` section, start the section with a level-two heading.

Here's an example of a Markdown file that uses the task template:

```
---
title: Configuring This Thing
---

{% capture overview %}
This page shows how to ...
{% endcapture %}

{% capture prerequisites %}
* Do this.
* Do this too.
{% endcapture %}

{% capture steps %}
## Doing ...

1. Do this.
1. Do this next. Possibly read this [related explanation](...).
{% endcapture %}

{% capture discussion %}
## Understanding ...

Here's an interesting thing to know about the steps you just did.
{% endcapture %}

{% capture whatsnext %}
* Learn more about [this](...).
* See this [related task](...).
{% endcapture %}

{% include templates/task.md %}
```

Here's an example of a published topic that uses the task template:

Using an HTTP Proxy to Access the Kubernetes API

Tutorial template

A tutorial page shows how to accomplish a goal that is larger than a single task. Typically a tutorial page has several sections, each of which has a sequence of steps. For example, a tutorial might provide a walkthrough of a code sample that illustrates a certain feature of Kubernetes. Tutorials can include surface-level explanations, but should link to related concept topics for deep explanations.

To write a new tutorial page, create a Markdown file in a subdirectory of the /docs/tutorials directory. In your Markdown file, provide values for these variables:

- overview - required
- prerequisites - required
- objectives - required
- lessoncontent - required
- cleanup - optional
- whatsnext - optional

Then include templates/tutorial.md like this:

```
...
{% include templates/tutorial.md %}
```

In the `lessoncontent` section, use `##` to start with a level-two heading. For subheadings, use `###` and `####` as needed.

Here's an example of a Markdown file that uses the tutorial template:

```
---
title: Running a Thing
---

{% capture overview %}
This page shows how to ...
{% endcapture %}

{% capture prerequisites %}
* Do this.
* Do this too.
{% endcapture %}

{% capture objectives %}
* Learn this.
* Build this.
* Run this.
{% endcapture %}

{% capture lessoncontent %}
## Building ...

1. Do this.
1. Do this next. Possibly read this [related explanation](...).

## Running ...

1. Do this.
1. Do this next.

## Understanding the code
Here's something interesting about the code you ran in the preceding steps.
{% endcapture %}

{% capture cleanup %}
* Delete this.
* Stop this.
{% endcapture %}

{% capture whatsnext %}
* Learn more about [this](...).
* See this [related tutorial](...).
{% endcapture %}

{% include templates/tutorial.md %}
```

Here's an example of a published topic that uses the tutorial template:

Running a Stateless Application Using a Deployment

Concept template

A concept page explains some aspect of Kubernetes. For example, a concept page might describe the Kubernetes Deployment object and explain the role it plays as an application is deployed, scaled, and updated. Typically, concept pages don't include sequences of steps, but instead provide links to tasks or tutorials.

To write a new concept page, create a Markdown file in a subdirectory of the /docs/concepts directory. In your Markdown file, provide values for these variables:

- overview - required
- body - required
- whatsnext - optional

Then include templates/concept.md like this:

```
...
{% include templates/concept.md %}
```

In the `body` section, use `##` to start with a level-two heading. For subheadings, use `###` and `####` as needed.

Here's an example of a page that uses the concept template:

```
---
title: Understanding this Thing
---

{% capture overview %}
This page explains ...
{% endcapture %}

{% capture body %}
## Understanding ...

Kubernetes provides ...

## Using ...

To use ...
{% endcapture %}

{% capture whatsnext %}
* Learn more about [this](...).
* See this [related task](...).
{% endcapture %}

{% include templates/concept.md %}
```

Here's an example of a published topic that uses the concept template:

Annotations

Reviewing Documentation Issues

This page explains how documentation issues are reviewed and prioritized for the kubernetes/kubernetes.github.io repository. The purpose is to provide a way to organize issues and make it easier to contribute to Kubernetes documentation. The following should be used as the standard way of prioritizing, labeling, and interacting with issues.

Categorizing issues

Issues should be sorted into different buckets of work using the following labels and definitions. If an issue doesn't have enough information to identify a problem that can be researched, reviewed, or worked on (i.e. the issue doesn't fit into any of the categories below) you should close the issue with a comment explaining why it is being closed.

Needs Clarification

- Issues that need more information from the original submitter to make them actionable. Issues with this label that aren't followed up within a week may be closed.

Actionable

- Issues that can be worked on with current information (or may need a comment to explain what needs to be done to make it more clear)
- Allows contributors to have easy to find issues to work on

Needs Tech Review

- Issues that need more information in order to be worked on (the proposed solution needs to be proven, a subject matter expert needs to be involved, work needs to be done to understand the problem/resolution and if the issue is still relevant)
- Promotes transparency about level of work needed for the issue and that issue is in progress

Needs Docs Review

- Issues that are suggestions for better processes or site improvements that require community agreement to be implemented
- Topics can be brought to SIG meetings as agenda items

Needs UX Review

- Issues that are suggestions for improving the user interface of the site.
- Fixing broken site elements.

Prioritizing Issues

The following labels and definitions should be used to prioritize issues. If you change the priority of an issues, please comment on the issue with your reasoning for the change.

P1

- Major content errors affecting more than 1 page
- Broken code sample on a heavily trafficked page
- Errors on a “getting started” page
- Well known or highly publicized customer pain points
- Automation issues

P2

- Default for all new issues
- Broken code for sample that is not heavily used
- Minor content issues in a heavily trafficked page
- Major content issues on a lower-trafficked page

P3

- Typos and broken anchor links

Handling special issue types

Duplicate issues

If a single problem has one or more issues open for it, the problem should be consolidated into a single issue. You should decide which issue to keep open (or open a new issue), port over all relevant information, link related issues, and close all the other issues that describe the same problem. Only having a single issue to work on will help reduce confusion and avoid duplicating work on the same problem.

Dead link issues

Depending on where the dead link is reported, different actions are required to resolve the issue. Dead links in the API and Kubectl docs are automation issues and should be assigned a P1 until the problem can be fully understood. All other dead links are issues that need to be manually fixed and can be assigned a P3.

What's next

- Learn about writing a new topic.
- Learn about using page templates.
- Learn about staging your changes.

Documentation Style Guide

This page gives writing style guidelines for the Kubernetes documentation. These are guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

For additional information on creating new content for the Kubernetes docs, follow the instructions on using page templates and creating a documentation pull request.

Documentation formatting standards

Use camel case for API objects

When you refer to an API object, use the same uppercase and lowercase letters that are used in the actual object name. Typically, the names of API objects use camel case.

Don’t split the API object name into separate words. For example, use PodTemplateList, not Pod Template List.

Refer to API objects without saying “object,” unless omitting “object” leads to an awkward construction.

Do	Don't
The Pod has two Containers.	The pod has two containers.
The Deployment is responsible for ...	The Deployment object is responsible for ...
A PodList is a list of Pods.	A Pod List is a list of pods.
The two ContainerPorts ...	The two ContainerPort objects ...
The two ContainerStateTerminated objects ...	The two ContainerStateTerminateds ...

Use angle brackets for placeholders

Use angle brackets for placeholders. Tell the reader what a placeholder represents.

1. Display information about a pod:

   ```
   kubectl describe pod <pod-name>
   ```

   where `<pod-name>` is the name of one of your pods.

Use bold for user interface elements

Do	Don't
Click **Fork**.	Click "Fork".
Select **Other**.	Select 'Other'.

Use italics to define or introduce new terms

Do	Don't
A *cluster* is a set of nodes ...	A "cluster" is a set of nodes ...
These components form the *control plane*.	These components form the **control plane.**

Use code style for filenames, directories, and paths

Do	Don't
Open the `envars.yaml` file.	Open the envars.yaml file.
Go to the `/docs/tutorials` directory.	Go to the /docs/tutorials directory.
Open the `/_data/concepts.yaml` file.	Open the /_data/concepts.yaml file.

Inline code formatting

Use code style for inline code and commands

For inline code in an HTML document, use the `<code>` tag. In a Markdown document, use the backtick (`).

Do	Don't
The `kubectl run` command creates a Deployment.	The "kubectl run" command creates a Deployment.
For declarative management, use `kubectl apply`.	For declarative management, use "kubectl apply".

Use code style for object field names

Do	Don't
Set the value of the `replicas` field in the configuration file.	Set the value of the "replicas" field in the configuration file.
The value of the `exec` field is an ExecAction object.	The value of the "exec" field is an ExecAction object.

Use normal style for string and integer field values

For field values of type string or integer, use normal style without quotation marks.

Do	Don't
Set the value of `imagePullPolicy` to Always.	Set the value of `imagePullPolicy` to "Always".
Set the value of `image` to nginx:1.8.	Set the value of `image` to `nginx:1.8`.
Set the value of the `replicas` field to 2.	Set the value of the `replicas` field to `2`.

Code snippet formatting

Don’t include the command prompt

Do	Don't
kubectl get pods	$ kubectl get pods

Separate commands from output

Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

```
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

Callout Formatting

Callouts help create different rhetorical appeal levels. Our documentation supports three different callouts: **Note:** {: .note}, **Caution:** {: .caution}, and **Warning:** {: .warning}.

1. Start each callout with the appropriate prefix.

2. Use the following syntax to apply a style:

```
**Note:** The prefix you use is the same text you use in the tag.
{: .note} <!-- This tag must appear on a new line. -->
```

The output is:

> **Note:** The prefix you choose is the same text for the tag.

Note

Use {: .note} to highlight a tip or a piece of information that may be helpful to know.

For example:

```
**Note:** You can _still_ use Markdown inside these callouts.
{: .note}
```

The output is:

> **Note:** You can *still* use Markdown inside these callouts.

Caution

Use {: .caution} to call attention to an important piece of information to avoid pitfalls.

For example:

```
**Caution:** The callout style only applies to the line directly above the tag.
{: .caution}
```

The output is:

> **Caution:** The callout style only applies to the line directly above the tag.

Warning

Use {: .warning} to indicate danger or a piece of information that is crucial to follow.

For example:

```
**Warning:** Beware.
{: .warning}
```

The output is:

> **Warning:** Beware.

Common Callout Issues

Style Does Not Apply

Callout tags must be on a new line to apply the style. Github's Preview Changes feature further obfuscates this fact by rendering the tag on the same line, but your code must match the following syntax:

```
**Note:** Your text goes here.
{: .note} <!-- This tag must appear on a new line. -->
```

Multiple Lines

Callouts automatically span multiple lines. However, you can use `<br/>` tags if you need to create multiple lines.

For example:

```
**Note:"** This is my note. Use `<br/>` to create multiple lines. <br/> <br/> You
{: .note}
```

The output is:

> **Note:** This is my note. Use `<br/>` to create multiple lines.
>
> You can still use *Markdown* to **format** text!

Typing multiple lines does **not** work. The callout style only applies to the line directly above the tag.

```
**Note:** This is my note.

I didn't read the style guide.
{: .note}
```

Note: This is my note.

> I didn't read the style guide.

Ordered Lists

Callouts will interrupt numbered lists unless you indent three spaces before the notice and the tag.

For example:

```
1. Preheat oven to 350˚F

1. Prepare the batter, and pour into springform pan.

   **Note:** Grease the pan for best results.
   {: .note}

1. Bake for 20-25 minutes or until set.
```

The output is:

1. Preheat oven to 350˚F
2. Prepare the batter, and pour into springform pan.

   > **Note:** Grease the pan for best results.

3. Bake for 20-25 minutes or until set.

Content best practices

This section contains suggested best practices for clear, concise, and consistent content.

Use present tense

Do	Don't
This command starts a proxy.	This command will start a proxy.

Exception: Use future or past tense if it is required to convey the correct meaning.

Use active voice

Do	Don't
You can explore the API using a browser.	The API can be explored using a browser.
The YAML file specifies the replica count.	The replica count is specified in the YAML file.

Exception: Use passive voice if active voice leads to an awkward construction.

Use simple and direct language

Use simple and direct language. Avoid using unnecessary phrases, such as saying “please.”

Do	Don't
To create a ReplicaSet, ...	In order to create a ReplicaSet, ...
See the configuration file.	Please see the configuration file.
View the Pods.	With this next command, we'll view the Pods.

Address the reader as “you”

Do	Don't
You can create a Deployment by ...	We'll create a Deployment by ...
In the preceding output, you can see...	In the preceding output, we can see ...

Avoid Latin phrases

Prefer English terms over Latin abbreviations.

Do	Don't
For example, ...	e.g., ...
That is, ...	i.e., ...

Exception: Use “etc.” for et cetera.

Patterns to avoid

Avoid using “we”

Using “we” in a sentence can be confusing, because the reader might not know whether they’re part of the “we” you’re describing.

Do	Don't
Version 1.4 includes ...	In version 1.4, we have added ...
Kubernetes provides a new feature for ...	We provide a new feature ...
This page teaches you how to use pods.	In this page, we are going to learn about pods.

Avoid jargon and idioms

Some readers speak English as a second language. Avoid jargon and idioms to help make their understanding easier.

Do	Don't
Internally, ...	Under the hood, ...
Create a new cluster.	Turn up a new cluster.

Avoid statements about the future

Avoid making promises or giving hints about the future. If you need to talk about an alpha feature, put the text under a heading that identifies it as alpha information.

Avoid statements that will soon be out of date

Avoid words like “currently” and “new.” A feature that is new today might not be considered new in a few months.

Do	Don't
In version 1.4, ...	In the current version, ...
The Federation feature provides ...	The new Federation feature provides ...

What’s next

- Learn about writing a new topic.

- Learn about using page templates.
- Learn about staging your changes
- Learn about creating a pull request.

Setup

This section provides instructions for installing Kubernetes and setting up a Kubernetes cluster. For an overview of the different options, see Picking the Right Solution.

Picking the Right Solution

Kubernetes can run on various platforms: from your laptop, to VMs on a cloud provider, to rack of bare metal servers. The effort required to set up a cluster varies from running a single command to crafting your own customized cluster. Use this guide to choose a solution that fits your needs.

If you just want to "kick the tires" on Kubernetes, use the local Docker-based solution using MiniKube.

When you are ready to scale up to more machines and higher availability, a hosted solution is the easiest to create and maintain.

Turnkey cloud solutions require only a few commands to create and cover a wide range of cloud providers.

If you already have a way to configure hosting resources, use kubeadm to easily bring up a cluster with a single command per machine.

Custom solutions vary from step-by-step instructions to general advice for setting up a Kubernetes cluster from scratch.

Local-machine Solutions

- Minikube is the recommended method for creating a local, single-node Kubernetes cluster for development and testing. Setup is completely automated and doesn't require a cloud provider account.

- Ubuntu on LXD supports a nine-instance deployment on localhost.
- IBM Cloud private-ce (Community Edition) can use VirtualBox on your machine to deploy Kubernetes to one or more VMs for dev and test scenarios. Scales to full multi-node cluster. Free version of the enterprise solution.

Hosted Solutions

- Google Container Engine offers managed Kubernetes clusters.
- Azure Container Service can easily deploy Kubernetes clusters.
- Stackpoint.io provides Kubernetes infrastructure automation and management for multiple public clouds.
- AppsCode.com provides managed Kubernetes clusters for various public clouds, including AWS and Google Cloud Platform.
- KUBE2GO.io get started with highly available Kubernetes clusters on multiple public clouds along with useful tools for development, debugging, monitoring.
- Madcore.Ai is devops-focused CLI tool for deploying Kubernetes infrastructure in AWS. Master, auto-scaling group nodes with spot-instances, ingress-ssl-lego, Heapster, and Grafana.
- Platform9 offers managed Kubernetes on-premises or on any public cloud, and provides 24/7 health monitoring and alerting.
- OpenShift Dedicated offers managed Kubernetes clusters powered by OpenShift.
- OpenShift Online provides free hosted access for Kubernetes applications.
- IBM Bluemix Container Service offers managed Kubernetes clusters with isolation choice, operational tools, integrated security insight into images and containers, and integration with Watson, IoT, and data.
- Giant Swarm offers managed Kubernetes clusters in their own datacenter, on-premises, or on public clouds.

Turnkey Cloud Solutions

These solutions allow you to create Kubernetes clusters on a range of Cloud IaaS providers with only a few commands. These solutions are actively developed and have active community support.

- Google Compute Engine (GCE)
- AWS
- Azure
- Tectonic by CoreOS
- CenturyLink Cloud
- IBM Bluemix
- Stackpoint.io
- KUBE2GO.io
- Madcore.Ai

Custom Solutions

Kubernetes can run on a wide range of Cloud providers and bare-metal environments, and with many base operating systems.

If you can find a guide below that matches your needs, use it. It may be a little out of date, but it will be easier than starting from scratch. If you do want to start from scratch, either because you have special requirements, or just because you want to understand what is underneath a Kubernetes cluster, try the Getting Started from Scratch guide.

If you are interested in supporting Kubernetes on a new platform, see Writing a Getting Started Guide.

Universal

If you already have a way to configure hosting resources, use kubeadm to easily bring up a cluster with a single command per machine.

Cloud

These solutions are combinations of cloud providers and operating systems not covered by the above solutions.

- CoreOS on AWS or GCE
- Kubernetes on Ubuntu
- Kubespray

On-Premises VMs

- Vagrant (uses CoreOS and flannel)
- CloudStack (uses Ansible, CoreOS and flannel)
- Vmware vSphere (uses Debian)
- Vmware Photon Controller (uses Debian)
- Vmware vSphere, OpenStack, or Bare Metal (uses Juju, Ubuntu and flannel)
- Vmware (uses CoreOS and flannel)
- CoreOS on libvirt (uses CoreOS)
- oVirt
- OpenStack Heat (uses CentOS and flannel)
- Fedora (Multi Node) (uses Fedora and flannel)

Bare Metal

- Offline (no internet required. Uses CoreOS and Flannel)
- Fedora via Ansible
- Fedora (Single Node)

- Fedora (Multi Node)
- CentOS
- Kubernetes on Ubuntu
- CoreOS on AWS or GCE

Integrations

These solutions provide integration with third-party schedulers, resource managers, and/or lower level platforms.

- Kubernetes on Mesos
  - Instructions specify GCE, but are generic enough to be adapted to most existing Mesos clusters
- DCOS
  - Community Edition DCOS uses AWS
  - Enterprise Edition DCOS supports cloud hosting, on-premises VMs, and bare metal

Table of Solutions

Below is a table of all of the solutions listed above.

IaaS Provider	Config. Mgmt.	OS	Networking	Docs	Support Level
any	any	multi-support	any CNI	docs	Project (SIG-cluster-lifecycle)
GKE			GCE	docs	Commercial
Stackpoint.io		multi-support	multi-support	docs	Commercial
AppsCode.com	Saltstack	Debian	multi-support	docs	Commercial
KUBE2GO.io		multi-support	multi-support	docs	Commercial
Madcore.Ai	Jenkins DSL	Ubuntu	flannel	docs	Community (@madcore-ai)
Platform9		multi-support	multi-support	docs	Commercial

IaaS Provider	Config. Mgmt.	OS	Networking	Docs	Support Level
Giant Swarm		CoreOS	flannel and/or Calico	docs	Commercial
GCE	Saltstack	Debian	GCE	docs	Project
Azure Container Service		Ubuntu	Azure	docs	Commercial
Azure (IaaS)		Ubuntu	Azure	docs	Community (Microsoft)
Bare-metal	Ansible	Fedora	flannel	docs	Project
Bare-metal	custom	Fedora	*none*	docs	Project
Bare-metal	custom	Fedora	flannel	docs	Community (@aveshagarwal)
libvirt	custom	Fedora	flannel	docs	Community (@aveshagarwal)
KVM	custom	Fedora	flannel	docs	Community (@aveshagarwal)
Mesos/Docker	custom	Ubuntu	Docker	docs	Community (Kubernetes-Mesos Authors)
Mesos/GCE				docs	Community (Kubernetes-Mesos Authors)
DCOS	Marathon	CoreOS/Alpine	custom	docs	Community (Kubernetes-Mesos Authors)
AWS	CoreOS	CoreOS	flannel	docs	Community
GCE	CoreOS	CoreOS	flannel	docs	Community (@pires)
Vagrant	CoreOS	CoreOS	flannel	docs	Community (@pires, @AntonioMeireles)
Bare-metal (Offline)	CoreOS	CoreOS	flannel	docs	Community (@jeffbean)
CloudStack	Ansible	CoreOS	flannel	docs	Community (@sebgoa)
Vmware vSphere	Saltstack	Debian	OVS	docs	Community (@imkin)
Vmware Photon	Saltstack	Debian	OVS	docs	Community (@alainroy)
Bare-metal	custom	CentOS	flannel	docs	Community (@coolsvap)
AWS	Juju	Ubuntu	flannel	docs	Commercial and Community (@matt, @chuck)
GCE	Juju	Ubuntu	flannel	docs	Commercial and Community (@matt, @chuck)
Bare Metal	Juju	Ubuntu	flannel	docs	Commercial and Community (@matt, @chuck)
Rackspace	Juju	Ubuntu	flannel	docs	Commercial and Community (@matt, @chuck)
Vmware vSphere	Juju	Ubuntu	flannel	docs	Commercial and Community (@matt, @chuck)
AWS	Saltstack	Debian	AWS	docs	Community (@justinsb)
AWS	kops	Debian	AWS	docs	Community (@justinsb)

IaaS Provider	Config. Mgmt.	OS	Networking	Docs	Support Level
Bare-metal	custom	Ubuntu	flannel	docs	Community (@resouer, @WIZARD-CXY)
libvirt/KVM	CoreOS	CoreOS	libvirt/KVM	docs	Community (@lhuard1A)
oVirt				docs	Community (@simon3z)
OpenStack Heat	Saltstack	CentOS	Neutron + flannel hostgw	docs	Community (@FujitsuEnablingSoftwareTechnologyGmbH)
any	any	any	any	docs	Community (@erictune)
any	any	any	any	docs	Commercial and Community

Note: The above table is ordered by version test/used in nodes, followed by support level.

Definition of columns

- **IaaS Provider** is the product or organization which provides the virtual or physical machines (nodes) that Kubernetes runs on.
- **OS** is the base operating system of the nodes.
- **Config. Mgmt.** is the configuration management system that helps install and maintain Kubernetes on the nodes.
- **Networking** is what implements the networking model. Those with networking type *none* may not support more than a single node, or may support multiple VM nodes in a single physical node.
- **Conformance** indicates whether a cluster created with this configuration has passed the project's conformance tests for supporting the API and base features of Kubernetes v1.0.0.
- **Support Levels**
  - **Project**: Kubernetes committers regularly use this configuration, so it usually works with the latest release of Kubernetes.
  - **Commercial**: A commercial offering with its own support arrangements.
  - **Community**: Actively supported by community contributions. May not work with recent releases of Kubernetes.

  - **Inactive**: Not actively maintained. Not recommended for first-time Kubernetes users, and may be removed.

- **Notes** has other relevant information, such as the version of Kubernetes used.

Running Kubernetes Locally via Minikube

Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a VM on your laptop for users looking to try out Kubernetes or develop with it day-to-day.

Minikube Features

- Minikube supports Kubernetes features such as:
  - DNS
  - NodePorts
  - ConfigMaps and Secrets
  - Dashboards
  - Container Runtime: Docker, and rkt
  - Enabling CNI (Container Network Interface)
  - Ingress

Installation

See Installing Minikube.

Quickstart

Here's a brief demo of minikube usage. If you want to change the VM driver add the appropriate `--vm-driver=xxx` flag to `minikube start`. Minikube supports the following drivers:

- virtualbox
- vmwarefusion
- kvm (driver installation)
- xhyve (driver installation)

Note that the IP below is dynamic and can change. It can be retrieved with `minikube ip`.

```
$ minikube start
Starting local Kubernetes cluster...
Running pre-create checks...
Creating machine...
Starting local Kubernetes cluster...

$ kubectl run hello-minikube --image=gcr.io/google_containers/echoserver:1.4 --por
deployment "hello-minikube" created
$ kubectl expose deployment hello-minikube --type=NodePort
service "hello-minikube" exposed

# We have now launched an echoserver pod but we have to wait until the pod is up b
# via the exposed service.
# To check whether the pod is up and running we can use the following:
$ kubectl get pod
NAME                              READY     STATUS              RESTARTS   AGE
hello-minikube-3383150820-vctvh   1/1       ContainerCreating   0          3s
# We can see that the pod is still being created from the ContainerCreating status
$ kubectl get pod
NAME                              READY     STATUS    RESTARTS   AGE
hello-minikube-3383150820-vctvh   1/1       Running   0          13s
# We can see that the pod is now Running and we will now be able to curl it:
$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=192.168.99.1
command=GET
real path=/
...
$ minikube stop
Stopping local Kubernetes cluster...
Stopping "minikube"...
```

Using rkt container engine

To use rkt as the container runtime run:

```
$ minikube start \
    --network-plugin=cni \
    --container-runtime=rkt \
    --iso-url=https://github.com/coreos/minikube-iso/releases/download/v0.0.5/mini
```

This will use an alternative minikube ISO image containing both rkt, and Docker, and enable CNI networking.

Driver plugins

See DRIVERS for details on supported drivers and how to install plugins, if required.

Reusing the Docker daemon

When using a single VM of Kubernetes, it's really handy to reuse the minikube's built-in Docker daemon; as this means you don't have to build a docker registry on your host machine and push the image into it - you can just build inside the same docker daemon as minikube which speeds up local experiments. Just make sure you tag your Docker image with something other than 'latest' and use that tag while you pull the image. Otherwise, if you do not specify version of your image, it will be assumed as `:latest`, with pull image policy of `Always` correspondingly, which may eventually result in `ErrImagePull` as you may not have any versions of your Docker image out there in the default docker registry (usually DockerHub) yet.

To be able to work with the docker daemon on your mac/linux host use the `docker-env command` in your shell:

```
eval $(minikube docker-env)
```

You should now be able to use docker on the command line on your host mac/linux machine talking to the docker daemon inside the minikube VM:

```
docker ps
```

On Centos 7, docker may report the following error:

```
Could not read CA certificate "/etc/docker/ca.pem": open /etc/docker/ca.pem: no su
```

The fix is to update /etc/sysconfig/docker to ensure that minikube's environment changes are respected:

```
< DOCKER_CERT_PATH=/etc/docker
---
> if [ -z "${DOCKER_CERT_PATH}" ]; then
>   DOCKER_CERT_PATH=/etc/docker
> fi
```

Remember to turn off the imagePullPolicy:Always, as otherwise Kubernetes won't use images you built locally.

Managing your Cluster

Starting a Cluster

The `minikube start` command can be used to start your cluster. This command creates and configures a virtual machine that runs a single-node Kubernetes cluster. This command also configures your kubectl installation to communicate with this cluster.

If you are behind a web proxy, you will need to pass this information in e.g. via

```
https_proxy=<my proxy> minikube start --docker-env HTTP_PROXY=<my proxy> --docker-
```

Unfortunately just setting the environment variables will not work.

Minikube will also create a "minikube" context, and set it to default in kubectl. To switch back to this context later, run this command: `kubectl config use-context minikube`.

Specifying the Kubernetes version

Minikube supports running multiple different versions of Kubernetes. You can access a list of all available versions via

```
minikube get-k8s-versions
```

You can specify the specific version of Kubernetes for Minikube to use by adding the `--kubernetes-version` string to the `minikube start` command. For example, to run version `v1.7.3`, you would run the following:

```
minikube start --kubernetes-version v1.7.3
```

Configuring Kubernetes

Minikube has a “configurator” feature that allows users to configure the Kubernetes components with arbitrary values. To use this feature, you can use the `--extra-config` flag on the `minikube start` command.

This flag is repeated, so you can pass it several times with several different values to set multiple options.

This flag takes a string of the form `component.key=value`, where `component` is one of the strings from the below list, `key` is a value on the configuration struct and `value` is the value to set.

Valid keys can be found by examining the documentation for the Kubernetes `componentconfigs` for each component. Here is the documentation for each supported configuration:

- kubelet
- apiserver
- proxy
- controller-manager
- etcd
- scheduler

Examples

To change the `MaxPods` setting to 5 on the Kubelet, pass this flag: `--extra-config=kubelet.MaxPods=5`.

This feature also supports nested structs. To change the `LeaderElection.LeaderElect` setting to `true` on the scheduler, pass this flag: `--extra-config=scheduler.LeaderElection.LeaderElect=true`.

To set the `AuthorizationMode` on the `apiserver` to `RBAC`, you can use: `--extra-config=apiserver.AuthorizationMode=RBAC`.

Stopping a Cluster

The `minikube stop` command can be used to stop your cluster. This command shuts down the minikube virtual machine, but preserves all cluster state and data. Starting the cluster again will restore it to it's previous state.

Deleting a Cluster

The `minikube delete` command can be used to delete your cluster. This command shuts down and deletes the minikube virtual machine. No data or state is preserved.

Interacting With your Cluster

Kubectl

The `minikube start` command creates a "kubectl context" called "minikube". This context contains the configuration to communicate with your minikube cluster.

Minikube sets this context to default automatically, but if you need to switch back to it in the future, run:

`kubectl config use-context minikube`,

Or pass the context on each command like this: `kubectl get pods --context=minikube`.

Dashboard

To access the Kubernetes Dashboard, run this command in a shell after starting minikube to get the address:

```
minikube dashboard
```

Services

To access a service exposed via a node port, run this command in a shell after starting minikube to get the address:

```
minikube service [-n NAMESPACE] [--url] NAME
```

Networking

The minikube VM is exposed to the host system via a host-only IP address, that can be obtained with the `minikube ip` command. Any services of type `NodePort` can be accessed over that IP address, on the NodePort.

To determine the NodePort for your service, you can use a `kubectl` command like this:

```
kubectl get service $SERVICE --output='jsonpath="{.spec.ports[0].nodePort}"'
```

Persistent Volumes

Minikube supports PersistentVolumes of type `hostPath`. These PersistentVolumes are mapped to a directory inside the minikube VM.

The Minikube VM boots into a tmpfs, so most directories will not be persisted across reboots (`minikube stop`). However, Minikube is configured to persist files stored under the following host directories:

- `/data`
- `/var/lib/localkube`
- `/var/lib/docker`

Here is an example PersistentVolume config to persist data in the `/data` directory:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 5Gi
  hostPath:
    path: /data/pv0001/
```

Mounted Host Folders

Some drivers will mount a host folder within the VM so that you can easily share files between the VM and host. These are not configurable at the moment and different for the driver and OS you are using.

Note: Host folder sharing is not implemented in the KVM driver yet.

Driver	OS	HostFolder	VM
VirtualBox	Linux	/home	/hosthome
VirtualBox	OSX	/Users	/Users
VirtualBox	Windows	C://Users	/c/Users
VMWare Fusion	OSX	/Users	/Users
Xhyve	OSX	/Users	/Users

Private Container Registries

To access a private container registry, follow the steps on this page.

We recommend you use `ImagePullSecrets`, but if you would like to configure access on the minikube VM you can place the `.dockercfg` in the `/home/docker` directory or the `config.json` in the `/home/docker/.docker` directory.

Add-ons

In order to have minikube properly start/restart custom addons, place the addons you wish to be launched with minikube in the `~/.minikube/addons` directory. Addons in this folder will be moved to the minikubeVM and launched each time minikube is started/restarted.

Using Minikube with an HTTP Proxy

Minikube creates a Virtual Machine that includes Kubernetes and a Docker daemon. When Kubernetes attempts to schedule containers using Docker, the Docker daemon may require external network access to pull containers.

If you are behind an HTTP proxy, you may need to supply Docker with the proxy settings. To do this, pass the required environment variables as flags during `minikube start`.

For example:

```
$ minikube start --docker-env HTTP_PROXY=http://$YOURPROXY:PORT \
                 --docker-env HTTPS_PROXY=https://$YOURPROXY:PORT
```

If your Virtual Machine address is 192.168.99.100, then chances are your proxy settings will prevent kubectl from directly reaching it. To by-pass proxy configuration for this IP address, you should modify your no_proxy settings. You can do so with:

```
$ export no_proxy=$no_proxy,$(minikube ip)
```

Known Issues

- Features that require a Cloud Provider will not work in Minikube. These include:
  - LoadBalancers
- Features that require multiple nodes. These include:
  - Advanced scheduling policies

Design

Minikube uses libmachine for provisioning VMs, and localkube (originally written and donated to this project by RedSpread) for running the cluster.

For more information about minikube, see the proposal.

Additional Links:

- **Goals and Non-Goals**: For the goals and non-goals of the minikube project, please see our roadmap.
- **Development Guide**: See CONTRIBUTING.md for an overview of how to send pull requests.
- **Building Minikube**: For instructions on how to build/test minikube from source, see the build guide
- **Adding a New Dependency**: For instructions on how to add a new dependency to minikube see the adding dependencies guide
- **Adding a New Addon**: For instruction on how to add a new addon for minikube see the adding an addon guide
- **Updating Kubernetes**: For instructions on how to update kubernetes see the updating Kubernetes guide

Community

Contributions, questions, and comments are all welcomed and encouraged! minikube developers hang out on Slack in the #minikube channel (get an invitation here). We also have the kubernetes-dev Google Groups mailing list. If you are posting to the list please prefix your subject with “minikube: “.

Using kubeadm to Create a Cluster

This quickstart shows you how to easily install a Kubernetes cluster on machines running Ubuntu 16.04+, CentOS 7 or HypriotOS v1.0.1+. The installation uses a tool called *kubeadm* which is part of Kubernetes. As of v1.6, kubeadm aims to create a secure cluster out of the box via mechanisms such as RBAC.

This process works with local VMs, physical servers and/or cloud servers. It is simple enough that you can easily integrate its use into your own automation (Terraform, Chef, Puppet, etc).

See the full kubeadm reference for information on all kubeadm command-line flags and for advice on automating kubeadm itself.

kubeadm assumes you have a set of machines (virtual or real) that are up and running. It is designed to be part of a large provisioning system - or just for easy manual provisioning. kubeadm is a great choice where you have your own infrastructure (e.g. bare metal), or where you have an existing orchestration system (e.g. Puppet) that you have to integrate with.

If you are not constrained, there are other higher-level tools built to give you complete clusters:

- On GCE, Google Container Engine gives you one-click Kubernetes clusters.
- On AWS, kops makes cluster installation and management easy. kops supports building high availability clusters (a feature that kubeadm is currently lacking but is building toward).

kubeadm Maturity

Aspect	Maturity Level
Command line UX	beta
Config file	alpha
Self-hosting	alpha
`kubeadm alpha` commands	alpha
Implementation	beta

The experience for the command line is currently in beta and we are trying hard not to change command line flags and break that flow. Other parts of the experience are still under active development. The implementation may change slightly as the tool evolves to support even easier upgrades and high availability (HA). Any commands under `kubeadm alpha` (not documented here) are, of course, alpha.

Be sure to read the limitations. Specifically, configuring cloud providers is difficult.

Before you begin

1. One or more machines running Ubuntu 16.04+, CentOS 7 or HypriotOS v1.0.1+
2. 1GB or more of RAM per machine (any less will leave little room for your apps)
3. Full network connectivity between all machines in the cluster (public or private network is fine)

Objectives

- Install a secure Kubernetes cluster on your machines
- Install a pod network on the cluster so that application components (pods) can talk to each other
- Install a sample microservices application (a socks shop) on the cluster

Instructions

(1/4) Installing kubeadm on your hosts

See Installing kubeadm.

Note: If you already have kubeadm installed, you should do a `apt-get update && apt-get upgrade` or `yum update` to get the latest version of kubeadm.

The kubelet is now restarting every few seconds, as it waits in a crashloop for kubeadm to tell it what to do.

(2/4) Initializing your master

The master is the machine where the control plane components run, including etcd (the cluster database) and the API server (which the kubectl CLI communicates with).

To initialize the master, pick one of the machines you previously installed kubeadm on, and run:

```
kubeadm init
```

Note:

- You need to choose a Pod Network Plugin in the next step. Depending on what third-party provider you choose, you might have to set the `--pod-network-cidr` to something provider-specific. The tabs below will contain a notice about what flags on `kubeadm init` are required.
- This will autodetect the network interface to advertise the master on as the interface with the default gateway. If you want to use a different interface, specify

`--apiserver-advertise-address=<ip-address>` argument to `kubeadm init`.

Please refer to the kubeadm reference doc if you want to read more about the flags `kubeadm init` provides.

`kubeadm init` will first run a series of prechecks to ensure that the machine is ready to run Kubernetes. It will expose warnings and exit on errors. It will then download and install the cluster database and control plane components. This may take several minutes.

You can't run `kubeadm init` twice without tearing down the cluster in between (unless you're upgrading from v1.6 to v1.7), see Tear Down.

The output should look like:

```
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production cluster
[init] Using Kubernetes version: v1.8.0
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks
[kubeadm] WARNING: starting in 1.8, tokens expire after 24 hours by default (if yo
[certificates] Generated ca certificate and key.
[certificates] Generated apiserver certificate and key.
[certificates] apiserver serving cert is signed for DNS names [kubeadm-master kube
[certificates] Generated apiserver-kubelet-client certificate and key.
[certificates] Generated sa key and public key.
[certificates] Generated front-proxy-ca certificate and key.
[certificates] Generated front-proxy-client certificate and key.
[certificates] Valid certificates and keys now exist in "/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "admin.conf"
[kubeconfig] Wrote KubeConfig file to disk: "kubelet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "controller-manager.conf"
[kubeconfig] Wrote KubeConfig file to disk: "scheduler.conf"
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kub
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to 
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kub
[etcd] Wrote Static Pod manifest for a local etcd instance to "/etc/kubernetes/man
[init] Waiting for the kubelet to boot up the control plane as Static Pods from di
[init] This often takes around a minute; or longer if the control plane images hav
[apiclient] All control plane components are healthy after 39.511972 seconds
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the
[markmaster] Will mark node master as master by adding a label and a taint
[markmaster] Master master tainted and labelled with key/value: node-role.kubernet
[bootstraptoken] Using token: <token>
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automat
[bootstraptoken] Creating the "cluster-info" ConfigMap in the "kube-public" namesp
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert
```

Make a record of the `kubeadm join` command that `kubeadm init` outputs. You will need this in a moment.

The token is used for mutual authentication between the master and the joining nodes. The token included here is secret, keep it safe — anyone with this token can add authenticated nodes to your cluster. These tokens can be listed, created and deleted with the `kubeadm token` command. See the reference guide.

(3/4) Installing a pod network

You **must** install a pod network add-on so that your pods can communicate with each other.

The network must be deployed before any applications. Also, kube-dns, a helper service, will not start up before a network is installed. kubeadm only supports Container Network Interface (CNI) based networks (and does not support kubenet).

Several projects provide Kubernetes pod networks using CNI, some of which also support Network Policy. See the add-ons page for a complete list of available network add-ons.

New for Kubernetes 1.6: kubeadm 1.6 sets up a more secure cluster by default. As such it uses RBAC to grant limited privileges to workloads running on the cluster. This includes networking integrations. As such, ensure that you are using a network system that has been updated to run with 1.6 and RBAC.

You can install a pod network add-on with the following command:

```
kubectl apply -f <add-on.yaml>
```

NOTE: You can install **only one** pod network per cluster.

Choose one... | Calico | Canal | Flannel | Kube-router | Romana | Weave Net

Please select one of the tabs to see installation instructions for the respective third-party Pod Network Provider.

Once a pod network has been installed, you can confirm that it is working by checking that the kube-dns pod is Running in the output of `kubectl get pods --all-namespaces`. And once the kube-dns pod is up and running, you can continue by joining your nodes.

If your network is not working or kube-dns is not in the Running state, check out the troubleshooting section below.

Master Isolation

By default, your cluster will not schedule pods on the master for security reasons. If you want to be able to schedule pods on the master, e.g. for a single-machine Kubernetes cluster for development, run:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

With output looking something like:

```
node "test-01" untainted
taint key="dedicated" and effect="" not found.
taint key="dedicated" and effect="" not found.
```

This will remove the `node-role.kubernetes.io/master` taint from any nodes that have it, including the master node, meaning that the scheduler will then be able to schedule pods everywhere.

(4/4) Joining your nodes

The nodes are where your workloads (containers and pods, etc) run. To add new nodes to your cluster do the following for each machine:

- SSH to the machine
- Become root (e.g. `sudo su -`)
- Run the command that was output by `kubeadm init`. For example:

```
kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-ce
```

The output should look something like:

```
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production cluster
[preflight] Running pre-flight checks
[discovery] Trying to connect to API Server "10.138.0.4:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://1
[discovery] Requesting info from "https://10.138.0.4:6443" again to validate TLS a
[discovery] Cluster info signature and contents are valid and TLS certificate vali
[discovery] Successfully established connection with API Server "10.138.0.4:6443"
[bootstrap] Detected server version: v1.8.0
[bootstrap] The server supports the Certificates API (certificates.k8s.io/v1beta1)
[csr] Created API client to obtain unique certificate for this node, generating ke
[csr] Received signed certificate from the API server, generating KubeConfig...

Node join complete:
* Certificate signing request sent to master and response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on the master to see this machine join.
```

A few seconds later, you should notice this node in the output from `kubectl get nodes` when run on the master.

(Optional) Controlling your cluster from machines other than the master

In order to get a kubectl on some other computer (e.g. laptop) to talk to your cluster, you need to copy the administrator kubeconfig file from your master to your workstation like this:

```
scp root@<master ip>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf get nodes
```

Note: If you are using GCE, instances disable ssh access for root by default. If that's the case you can log in to the machine, copy the file someplace that can be accessed and then use `gcloud compute copy-files`.

(Optional) Proxying API Server to localhost

If you want to connect to the API Server from outside the cluster you can use `kubectl proxy`:

```
scp root@<master ip>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf proxy
```

You can now access the API Server locally at `http://localhost:8001/api/v1`

(Optional) Installing a sample application

Now it is time to take your new cluster for a test drive. Sock Shop is a sample microservices application that shows how to run and connect a set of services on Kubernetes. To learn more about the sample microservices app, see the GitHub README.

Note that the Sock Shop demo only works on `amd64`.

```
kubectl create namespace sock-shop
kubectl apply -n sock-shop -f "https://github.com/microservices-demo/microservices
```

You can then find out the port that the NodePort feature of services allocated for the front-end service by running:

```
kubectl -n sock-shop get svc front-end
```

Sample output:

```
NAME        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
front-end   10.110.250.153  <nodes>       80:30001/TCP   59s
```

It takes several minutes to download and start all the containers, watch the output of `kubectl get pods -n sock-shop` to see when they're all up and running.

Then go to the IP address of your cluster's master node in your browser, and specify the given port. So for example, `http://<master_ip>:<port>`. In the example above, this was `30001`, but it may be a different port for you.

If there is a firewall, make sure it exposes this port to the internet before you try to access it.

To uninstall the socks shop, run `kubectl delete namespace sock-shop` on the master.

Tear down

To undo what kubeadm did, you should first drain the node and make sure that the node is empty before shutting it down.

Talking to the master with the appropriate credentials, run:

```
kubectl drain <node name> --delete-local-data --force --ignore-daemonsets
kubectl delete node <node name>
```

Then, on the node being removed, reset all kubeadm installed state:

```
kubeadm reset
```

If you wish to start over simply run `kubeadm init` or `kubeadm join` with the appropriate arguments.

Upgrading

Instructions for upgrading kubeadm clusters are available for:

- 1.6 to 1.7 upgrades
- 1.7.x to 1.7.y upgrades
- 1.7 to 1.8 upgrades
- 1.8.x to 1.8.y upgrades

Explore other add-ons

See the list of add-ons to explore other add-ons, including tools for logging, monitoring, network policy, visualization & control of your Kubernetes cluster.

What’s next

- Learn about kubeadm’s advanced usage on the advanced reference doc.
- Learn more about Kubernetes concepts and `kubectl`.
- Configure log rotation. You can use **logrotate** for that. When using Docker, you can specify log rotation options for Docker daemon, for example `--log-driver=json-file --log-opt=max-size=10m --log-opt=max-file=5`. See Configure and troubleshoot the Docker daemon for more details.

Feedback

- kubeadm support Slack Channel: kubeadm
- General SIG Cluster Lifecycle Development Slack Channel: sig-cluster-lifecycle
- Mailing List: kubernetes-sig-cluster-lifecycle
- GitHub Issues in the kubeadm repository

Version skew policy

The kubeadm CLI tool of version vX.Y may deploy clusters with a control plane of version vX.Y or vX.(Y-1). kubeadm CLI vX.Y can also upgrade an existing kubeadm-created cluster of version vX.(Y-1).

Due to that we can’t see into the future, kubeadm CLI vX.Y may or may not be able to deploy vX.(Y+1) clusters.

Example: kubeadm v1.8 can deploy both v1.7 and v1.8 clusters and upgrade v1.7 kubeadm-created clusters to v1.8.

kubeadm is multi-platform

kubeadm deb/rpm packages and binaries are built for amd64, arm (32-bit), arm64, ppc64le, and s390x following the multi-platform proposal.

Only some of the network providers offer solutions for all platforms. Please consult the list of network providers above or the documentation from each provider to figure out whether the provider supports your chosen platform.

Limitations

Please note: kubeadm is a work in progress and these limitations will be addressed in due course.

1. The cluster created here has a single master, with a single etcd database running on it. This means that if the master fails, your cluster loses its configuration data and will need to be recreated from scratch. Adding HA support (multiple etcd servers, multiple API servers, etc) to kubeadm is still a work-in-progress.

   Workaround: regularly back up etcd. The etcd data directory configured by kubeadm is at `/var/lib/etcd` on the master.

Troubleshooting

You may have trouble in the configuration if you see Pod statuses like `RunContainerError`, `CrashLoopBackOff` or `Error`.

1. **There are Pods in the `RunContainerError`, `CrashLoopBackOff` or `Error` state**. Right after `kubeadm init` there should not be any such Pods. If there are Pods in such a state *right after* `kubeadm init`, please open an issue in the kubeadm repo. `kube-dns` should be in the `Pending` state until you have deployed the network solution. However, if you see Pods in the `RunContainerError`, `CrashLoopBackOff` or `Error` state after deploying the network solution and nothing happens to `kube-dns`, it's very likely that the Pod Network solution that you installed is somehow broken. You might have to grant it more RBAC privileges or use a newer version. Please file an issue in the Pod Network providers' issue tracker and get the issue triaged there.

2. **The `kube-dns` Pod is stuck in the `Pending` state forever**. This is expected and part of the design. kubeadm is network provider-agnostic, so the admin should install the pod network solution of choice. You have to install a Pod Network before `kube-dns` may deployed fully. Hence the `Pending` state before the network is set up.

3. **I tried to set `HostPort` on one workload, but it didn't have any effect**. The `HostPort` and `HostIP` functionality is available depending on your Pod Network provider. Please contact the author of the Pod Network solution to find out whether `HostPort` and `HostIP` functionality are available.

   If not, you may still use the NodePort feature of services or use `HostNetwork=true`.

4. **Pods cannot access themselves via their Service IP**. Many network add-ons do not yet enable hairpin mode which allows pods to access themselves via their Service IP if they don't know about their podIP. This is an issue related to CNI. Please contact the providers of the network add-on providers to get timely information about whether they support hairpin mode.

5. If you are using VirtualBox (directly or via Vagrant), you will need to ensure that `hostname -i` returns a routable IP address (i.e. one on the second network interface, not the first one). By default, it doesn't do this and kubelet ends-up using first non-loopback network interface, which is usually NATed. Workaround: Modify `/etc/hosts`, take a look at this `Vagrantfile` ubuntu-vagrantfile for how this can be achieved.

6. The following error indicates a possible certificate mismatch.

```
# kubectl get po
Unable to connect to the server: x509: certificate signed by unknown authority (po
```

Verify that the `$HOME/.kube/config` file contains a valid certificate, and regenerate a certificate if necessary. Another workaround is to overwrite the default `kubeconfig` for the "admin" user:

```
mv  $HOME/.kube $HOME/.kube.bak
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

1. If you are using CentOS and encounter difficulty while setting up the master node， verify that your Docker cgroup driver matches the kubelet config:

```
docker info |grep -i cgroup
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

If the Docker cgroup driver and the kubelet config don’t match, change the kubelet config to match the Docker cgroup driver.

Update

```
KUBELET_CGROUP_ARGS=--cgroup-driver=systemd
```

To

```
KUBELET_CGROUP_ARGS=--cgroup-driver=cgroupfs
```

Then restart kubelet:

```
systemctl daemon-reload
systemctl restart kubelet
```

The `kubectl describe pod` or `kubectl logs` commands can help you diagnose errors. For example:

```
kubectl -n ${NAMESPACE} describe pod ${POD_NAME}

kubectl -n ${NAMESPACE} logs ${POD_NAME} -c ${CONTAINER_NAME}
```

Creating a Custom Cluster from Scratch

This guide is for people who want to craft a custom Kubernetes cluster. If you can find an existing Getting Started Guide that meets your needs on this list, then we recommend using it, as you will be able to benefit from the experience of others. However, if you have specific IaaS, networking, configuration management, or operating system requirements not met by any of those guides, then this guide will provide an outline of the steps you need to take. Note that it requires considerably more effort than using one of the pre-defined guides.

This guide is also useful for those wanting to understand at a high level some of the steps that existing cluster setup scripts are making.

Designing and Preparing

Learning

1. You should be familiar with using Kubernetes already. We suggest you set up a temporary cluster by following one of the other Getting Started Guides. This will help you become familiar with the CLI (kubectl) and concepts (pods, services, etc.) first.
2. You should have `kubectl` installed on your desktop. This will happen as a side effect of completing one of the other Getting Started Guides. If not, follow the instructions here.

Cloud Provider

Kubernetes has the concept of a Cloud Provider, which is a module which provides an interface for managing TCP Load Balancers, Nodes (Instances) and Networking Routes. The interface is defined in `pkg/cloudprovider/cloud.go`. It is possible to create a custom cluster without implementing a cloud provider (for example if using bare-metal), and not all parts of the interface need to be implemented, depending on how flags are set on various components.

Nodes

- You can use virtual or physical machines.
- While you can build a cluster with 1 machine, in order to run all the examples and tests you need at least 4 nodes.
- Many Getting-started-guides make a distinction between the master node and regular nodes. This is not strictly necessary.
- Nodes will need to run some version of Linux with the x86_64 architecture. It may be possible to run on other OSes and Architectures, but this guide does not try to assist with that.
- Apiserver and etcd together are fine on a machine with 1 core and 1GB RAM for clusters with 10s of nodes. Larger or more active clusters may benefit from more cores.
- Other nodes can have any reasonable amount of memory and any number of cores. They need not have identical configurations.

Network

Network Connectivity

Kubernetes has a distinctive networking model.

Kubernetes allocates an IP address to each pod. When creating a cluster, you need to allocate a block of IPs for Kubernetes to use as Pod IPs. The simplest approach is to allocate a different block of IPs to each node in the cluster as the node is added. A process in one pod should be able to communicate with another pod using the IP of the second pod. This connectivity can be accomplished in two ways:

- **Using an overlay network**
  - An overlay network obscures the underlying network architecture from the pod network through traffic encapsulation (e.g. vxlan).
  - Encapsulation reduces performance, though exactly how much depends on your solution.
- **Without an overlay network**
  - Configure the underlying network fabric (switches, routers, etc.) to be aware of pod IP addresses.

- This does not require the encapsulation provided by an overlay, and so can achieve better performance.

Which method you choose depends on your environment and requirements. There are various ways to implement one of the above options:

- **Use a network plugin which is called by Kubernetes**
  - Kubernetes supports the CNI network plugin interface.
  - There are a number of solutions which provide plugins for Kubernetes (listed alphabetically):
    - Calico
    - Flannel
    - Open vSwitch (OVS)
    - Romana
    - Weave
    - More found here
  - You can also write your own.
- **Compile support directly into Kubernetes**
  - This can be done by implementing the "Routes" interface of a Cloud Provider module.
  - The Google Compute Engine (GCE/) and AWS guides use this approach.
- **Configure the network external to Kubernetes**
  - This can be done by manually running commands, or through a set of externally maintained scripts.
  - You have to implement this yourself, but it can give you an extra degree of flexibility.

You will need to select an address range for the Pod IPs. Note that IPv6 is not yet supported for Pod IPs.

- Various approaches:

  - GCE: each project has its own `10.0.0.0/8`. Carve off a `/16` for each Kubernetes cluster from that space, which leaves room for several clusters. Each node gets a further subdivision of this space.
  - AWS: use one VPC for whole organization, carve off a chunk for each cluster, or use different VPC for different clusters.
- Allocate one CIDR subnet for each node's PodIPs, or a single large CIDR from which smaller CIDRs are automatically allocated to each node.
  - You need max-pods-per-node * max-number-of-nodes IPs in total. A `/24` per node supports 254 pods per machine and is a common choice. If IPs are scarce, a `/26` (62 pods per machine) or even a `/27` (30 pods) may be sufficient.
  - e.g. use `10.10.0.0/16` as the range for the cluster, with up to 256 nodes using `10.10.0.0/24` through `10.10.255.0/24`, respectively.
  - Need to make these routable or connect with overlay.

Kubernetes also allocates an IP to each service. However, service IPs do not necessarily need to be routable. The kube-proxy takes care of translating Service IPs to Pod IPs before traffic leaves the node. You do need to Allocate a block of IPs for services. Call this `SERVICE_CLUSTER_IP_RANGE`. For example, you could set `SERVICE_CLUSTER_IP_RANGE="10.0.0.0/16"`, allowing 65534 distinct services to be active at once. Note that you can grow the end of this range, but you cannot move it without disrupting the services and pods that already use it.

Also, you need to pick a static IP for master node.

- Call this `MASTER_IP`.
- Open any firewalls to allow access to the apiserver ports 80 and/or 443.
- Enable ipv4 forwarding sysctl, `net.ipv4.ip_forward = 1`

Network Policy

Kubernetes enables the definition of fine-grained network policy between Pods using the NetworkPolicy resource.

Not all networking providers support the Kubernetes NetworkPolicy API, see Using Network Policy for more information.

Cluster Naming

You should pick a name for your cluster. Pick a short name for each cluster which is unique from future cluster names. This will be used in several ways:

- by kubectl to distinguish between various clusters you have access to. You will probably want a second one sometime later, such as for testing new Kubernetes releases, running in a different region of the world, etc.
- Kubernetes clusters can create cloud provider resources (e.g. AWS ELBs) and different clusters need to distinguish which resources each created. Call this `CLUSTER_NAME` .

Software Binaries

You will need binaries for:

- etcd
- A container runner, one of:
  - docker
  - rkt
- Kubernetes
  - kubelet
  - kube-proxy
  - kube-apiserver
  - kube-controller-manager
  - kube-scheduler

Downloading and Extracting Kubernetes Binaries

A Kubernetes binary release includes all the Kubernetes binaries as well as the supported release of etcd. You can use a Kubernetes binary release (recommended) or build your Kubernetes binaries following the instructions in the Developer Documentation. Only using a binary release is covered in this guide.

Download the latest binary release and unzip it. Server binary tarballs are no longer included in the Kubernetes final tarball, so you will need to locate and run `./kubernetes/cluster/get-kube-binaries.sh` to download the client and server binaries. Then locate `./kubernetes/server/kubernetes-server-linux-amd64.tar.gz` and unzip *that*. Then, within the second set of unzipped files, locate `./kubernetes/server/bin`, which contains all the necessary binaries.

Selecting Images

You will run docker, kubelet, and kube-proxy outside of a container, the same way you would run any system daemon, so you just need the bare binaries. For etcd, kube-apiserver, kube-controller-manager, and kube-scheduler, we recommend that you run these as containers, so you need an image to be built.

You have several choices for Kubernetes images:

- Use images hosted on Google Container Registry (GCR):
  - e.g. `gcr.io/google_containers/hyperkube:$TAG`, where `TAG` is the latest release tag, which can be found on the latest releases page.
  - Ensure $TAG is the same tag as the release tag you are using for kubelet and kube-proxy.
  - The hyperkube binary is an all in one binary
    - `hyperkube kubelet ...` runs the kubelet, `hyperkube apiserver ...` runs an apiserver, etc.
- Build your own images.
  - Useful if you are using a private registry.
  - The release contains files such as `./kubernetes/server/bin/kube-apiserver.tar` which can be converted into docker images using a command like `docker load -i kube-apiserver.tar`

- You can verify if the image is loaded successfully with the right repository and tag using command like `docker images`

For etcd, you can:

- Use images hosted on Google Container Registry (GCR), such as `gcr.io/google_containers/etcd:2.2.1`
- Use images hosted on Docker Hub or Quay.io, such as `quay.io/coreos/etcd:v2.2.1`
- Use etcd binary included in your OS distro.
- Build your own image
  - You can do: `cd kubernetes/cluster/images/etcd; make`

We recommend that you use the etcd version which is provided in the Kubernetes binary distribution. The Kubernetes binaries in the release were tested extensively with this version of etcd and not with any other version. The recommended version number can also be found as the value of `TAG` in `kubernetes/cluster/images/etcd/Makefile`.

The remainder of the document assumes that the image identifiers have been chosen and stored in corresponding env vars. Examples (replace with latest tags and appropriate registry):

- `HYPERKUBE_IMAGE=gcr.io/google_containers/hyperkube:$TAG`
- `ETCD_IMAGE=gcr.io/google_containers/etcd:$ETCD_VERSION`

Security Models

There are two main options for security:

- Access the apiserver using HTTP.
  - Use a firewall for security.
  - This is easier to setup.
- Access the apiserver using HTTPS
  - Use https with certs, and credentials for user.

  - This is the recommended approach.
  - Configuring certs can be tricky.

If following the HTTPS approach, you will need to prepare certs and credentials.

Preparing Certs

You need to prepare several certs:

- The master needs a cert to act as an HTTPS server.
- The kubelets optionally need certs to identify themselves as clients of the master, and when serving its own API over HTTPS.

Unless you plan to have a real CA generate your certs, you will need to generate a root cert and use that to sign the master, kubelet, and kubectl certs. How to do this is described in the authentication documentation.

You will end up with the following files (we will use these variables later on)

- `CA_CERT`
  - put in on node where apiserver runs, in e.g. `/srv/kubernetes/ca.crt`.
- `MASTER_CERT`
  - signed by CA_CERT
  - put in on node where apiserver runs, in e.g. `/srv/kubernetes/server.crt`
- `MASTER_KEY`
  - put in on node where apiserver runs, in e.g. `/srv/kubernetes/server.key`
- `KUBELET_CERT`
  - optional
- `KUBELET_KEY`
  - optional

Preparing Credentials

The admin user (and any users) need:

- a token or a password to identify them.
- tokens are just long alphanumeric strings, e.g. 32 chars. See

  -
    ```
    TOKEN=$(dd if=/dev/urandom bs=128 count=1 2>/dev/null | base64 | tr -d "=+/" | dd bs=32 count=1 2>/dev/null)
    ```

Your tokens and passwords need to be stored in a file for the apiserver to read. This guide uses `/var/lib/kube-apiserver/known_tokens.csv`. The format for this file is described in the authentication documentation.

For distributing credentials to clients, the convention in Kubernetes is to put the credentials into a kubeconfig file.

The kubeconfig file for the administrator can be created as follows:

- If you have already used Kubernetes with a non-custom cluster (for example, used a Getting Started Guide), you will already have a `$HOME/.kube/config` file.
- You need to add certs, keys, and the master IP to the kubeconfig file:
  - If using the firewall-only security option, set the apiserver this way:

    -
      ```
      kubectl config set-cluster $CLUSTER_NAME --server=http://$MASTER_IP --insecure-skip-tls-verify=true
      ```
  - Otherwise, do this to set the apiserver ip, client certs, and user credentials.

    -
      ```
      kubectl config set-cluster $CLUSTER_NAME --certificate-authority=$CA_CERT --embed-certs=true --server=https://$MASTER_IP
      ```

    -
      ```
      kubectl config set-credentials $USER --client-certificate=$CLI_CERT --client-key=$CLI_KEY --embed-certs=true --token=$TOKEN
      ```
  - Set your cluster as the default cluster to use:

    -
      ```
      kubectl config set-context $CONTEXT_NAME --cluster=$CLUSTER_NAME --user=$USER
      ```

- `kubectl config use-context $CONTEXT_NAME`

Next, make a kubeconfig file for the kubelets and kube-proxy. There are a couple of options for how many distinct files to make:

1. Use the same credential as the admin - This is simplest to setup.
2. One token and kubeconfig file for all kubelets, one for all kube-proxy, one for admin. - This mirrors what is done on GCE today
3. Different credentials for every kubelet, etc. - We are working on this but all the pieces are not ready yet.

You can make the files by copying the `$HOME/.kube/config`, by following the code in `cluster/gce/configure-vm.sh` or by using the following template:

```
apiVersion: v1
kind: Config
users:
- name: kubelet
  user:
    token: ${KUBELET_TOKEN}
clusters:
- name: local
  cluster:
    certificate-authority: /srv/kubernetes/ca.crt
contexts:
- context:
    cluster: local
    user: kubelet
  name: service-account-context
current-context: service-account-context
```

Put the kubeconfig(s) on every node. The examples later in this guide assume that there are kubeconfigs in `/var/lib/kube-proxy/kubeconfig` and `/var/lib/kubelet/kubeconfig`.

Configuring and Installing Base Software on Nodes

This section discusses how to configure machines to be Kubernetes nodes.

You should run three daemons on every node:

- docker or rkt
- kubelet
- kube-proxy

You will also need to do assorted other configuration on top of a base OS install.

Tip: One possible starting point is to setup a cluster using an existing Getting Started Guide. After getting a cluster running, you can then copy the init.d scripts or systemd unit files from that cluster, and then modify them for use on your custom cluster.

Docker

The minimum required Docker version will vary as the kubelet version changes. The newest stable release is a good choice. Kubelet will log a warning and refuse to start pods if the version is too old, so pick a version and try it.

If you previously had Docker installed on a node without setting Kubernetes-specific options, you may have a Docker-created bridge and iptables rules. You may want to remove these as follows before proceeding to configure Docker for Kubernetes.

```
iptables -t nat -F
ip link set docker0 down
ip link delete docker0
```

The way you configure docker will depend in whether you have chosen the routable-vip or overlay-network approaches for your network. Some suggested docker options:

- create your own bridge for the per-node CIDR ranges, call it cbr0, and set `--bridge=cbr0` option on docker.
- set `--iptables=false` so docker will not manipulate iptables for host-ports (too coarse on older docker versions, may be fixed in newer versions) so that kube-proxy can manage iptables instead of docker.
- `--ip-masq=false`
    - if you have setup PodIPs to be routable, then you want this false, otherwise, docker will rewrite the PodIP source-address to a NodeIP.

    - some environments (e.g. GCE) still need you to masquerade out-bound traffic when it leaves the cloud environment. This is very environment specific.
    - if you are using an overlay network, consult those instructions.
- `--mtu=`
    - may be required when using Flannel, because of the extra packet size due to udp encapsulation
- `--insecure-registry $CLUSTER_SUBNET`
    - to connect to a private registry, if you set one up, without using SSL.

You may want to increase the number of open files for docker:

- `DOCKER_NOFILE=1000000`

Where this config goes depends on your node OS. For example, GCE's Debian-based distro uses `/etc/default/docker` .

Ensure docker is working correctly on your system before proceeding with the rest of the installation, by following examples given in the Docker documentation.

rkt

rkt is an alternative to Docker. You only need to install one of Docker or rkt. The minimum version required is v0.5.6.

systemd is required on your node to run rkt. The minimum version required to match rkt v0.5.6 is systemd 215.

rkt metadata service is also required for rkt networking support. You can start rkt metadata service by using command like `sudo systemd-run rkt metadata-service`

Then you need to configure your kubelet with flag:

- `--container-runtime=rkt`

kubelet

All nodes should run kubelet. See Software Binaries.

Arguments to consider:

- If following the HTTPS security approach:
  - `--api-servers=https://$MASTER_IP`
  - `--kubeconfig=/var/lib/kubelet/kubeconfig`
- Otherwise, if taking the firewall-based security approach
  - `--api-servers=http://$MASTER_IP`
- `--config=/etc/kubernetes/manifests`
- `--cluster-dns=` to the address of the DNS server you will setup (see Starting Cluster Services.)
- `--cluster-domain=` to the dns domain prefix to use for cluster DNS addresses.
- `--docker-root=`
- `--root-dir=`
- `--configure-cbr0=` (described below)
- `--register-node` (described in Node documentation.)

kube-proxy

All nodes should run kube-proxy. (Running kube-proxy on a "master" node is not strictly required, but being consistent is easier.) Obtain a binary as described for kubelet.

Arguments to consider:

- If following the HTTPS security approach:
  - `--master=https://$MASTER_IP`
  - `--kubeconfig=/var/lib/kube-proxy/kubeconfig`
- Otherwise, if taking the firewall-based security approach

- `--master=http://$MASTER_IP`

Networking

Each node needs to be allocated its own CIDR range for pod networking. Call this `NODE_X_POD_CIDR`
.

A bridge called `cbr0` needs to be created on each node. The bridge is explained further in the networking documentation. The bridge itself needs an address from `$NODE_X_POD_CIDR` - by convention the first IP. Call this `NODE_X_BRIDGE_ADDR`. For example, if `NODE_X_POD_CIDR` is `10.0.0.0/16`, then `NODE_X_BRIDGE_ADDR` is `10.0.0.1/16`. NOTE: this retains the `/16` suffix because of how this is used later.

- Recommended, automatic approach:
  1. Set `--configure-cbr0=true` option in kubelet init script and restart kubelet service. Kubelet will configure cbr0 automatically. It will wait to do this until the node controller has set Node.Spec.PodCIDR. Since you have not setup apiserver and node controller yet, the bridge will not be setup immediately.
- Alternate, manual approach:
  1. Set `--configure-cbr0=false` on kubelet and restart.
  2. Create a bridge.

     ```
     ip link add name cbr0 type bridge
     ```
  3. Set appropriate MTU. NOTE: the actual value of MTU will depend on your network environment

     ```
     ip link set dev cbr0 mtu 1460
     ```
  4. Add the node’s network to the bridge (docker will go on other side of bridge).

     ```
     ip addr add $NODE_X_BRIDGE_ADDR dev cbr0
     ```
  5. Turn it on

     ```
     ip link set dev cbr0 up
     ```

If you have turned off Docker's IP masquerading to allow pods to talk to each other, then you may need to do masquerading just for destination IPs outside the cluster network. For example:

```
iptables -t nat -A POSTROUTING ! -d ${CLUSTER_SUBNET} -m addrtype ! --dst-type LOC
```

This will rewrite the source address from the PodIP to the Node IP for traffic bound outside the cluster, and kernel connection tracking will ensure that responses destined to the node still reach the pod.

NOTE: This is environment specific. Some environments will not need any masquerading at all. Others, such as GCE, will not allow pod IPs to send traffic to the internet, but have no problem with them inside your GCE Project.

Other

- Enable auto-upgrades for your OS package manager, if desired.
- Configure log rotation for all node components (e.g. using logrotate).
- Setup liveness-monitoring (e.g. using supervisord).
- Setup volume plugin support (optional)
  - Install any client binaries for optional volume types, such as `glusterfs-client` for GlusterFS volumes.

Using Configuration Management

The previous steps all involved "conventional" system administration techniques for setting up machines. You may want to use a Configuration Management system to automate the node configuration process. There are examples of Saltstack, Ansible, Juju, and CoreOS Cloud Config in the various Getting Started Guides.

Bootstrapping the Cluster

While the basic node services (kubelet, kube-proxy, docker) are typically started and managed using traditional system administration/automation approaches, the remaining *master* components of Kubernetes are all configured and managed *by Kubernetes*:

- Their options are specified in a Pod spec (yaml or json) rather than an /etc/init.d file or systemd unit.
- They are kept running by Kubernetes rather than by init.

etcd

You will need to run one or more instances of etcd.

- Highly available and easy to restore - Run 3 or 5 etcd instances with, their logs written to a directory backed by durable storage (RAID, GCE PD)
- Not highly available, but easy to restore - Run one etcd instance, with its log written to a directory backed by durable storage (RAID, GCE PD) **Note:** May result in operations outages in case of instance outage
- Highly available - Run 3 or 5 etcd instances with non durable storage. **Note:** Log can be written to non-durable storage because storage is replicated.

See cluster-troubleshooting for more discussion on factors affecting cluster availability.

To run an etcd instance:

1. Copy `cluster/saltbase/salt/etcd/etcd.manifest`
2. Make any modifications needed
3. Start the pod by putting it into the kubelet manifest directory

Apiserver, Controller Manager, and Scheduler

The apiserver, controller manager, and scheduler will each run as a pod on the master node.

For each of these components, the steps to start them running are similar:

1. Start with a provided template for a pod.

2. Set the `HYPERKUBE_IMAGE` to the values chosen in Selecting Images.
3. Determine which flags are needed for your cluster, using the advice below each template.
4. Set the flags to be individual strings in the command array (e.g. $ARGN below)
5. Start the pod by putting the completed template into the kubelet manifest directory.
6. Verify that the pod is started.

Apiserver pod template

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-apiserver"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-apiserver",
        "image": "${HYPERKUBE_IMAGE}",
        "command": [
          "/hyperkube",
          "apiserver",
          "$ARG1",
          "$ARG2",
          ...
          "$ARGN"
        ],
        "ports": [
          {
            "name": "https",
            "hostPort": 443,
            "containerPort": 443
          },
          {
            "name": "local",
            "hostPort": 8080,
            "containerPort": 8080
          }
        ],
        "volumeMounts": [
          {
            "name": "srvkube",
            "mountPath": "/srv/kubernetes",
            "readOnly": true
          },
          {
            "name": "etcssl",
            "mountPath": "/etc/ssl",
            "readOnly": true
          }
        ],
        "livenessProbe": {
          "httpGet": {
            "scheme": "HTTP",
            "host": "127.0.0.1",
            "port": 8080,
            "path": "/healthz"
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ],
    "volumes": [
      {
        "name": "srvkube",
        "hostPath": {
          "path": "/srv/kubernetes"
        }
      },
      {
        "name": "etcssl",
        "hostPath": {
          "path": "/etc/ssl"
        }
      }
    ]
  }
}
```

Here are some apiserver flags you may need to set:

- `--cloud-provider=` see cloud providers
- `--cloud-config=` see cloud providers
- `--address=${MASTER_IP}` *or* `--bind-address=127.0.0.1` and `--address=127.0.0.1` if you want to run a proxy on the master node.
- `--service-cluster-ip-range=$SERVICE_CLUSTER_IP_RANGE`

- `--etcd-servers=http://127.0.0.1:4001`
- `--tls-cert-file=/srv/kubernetes/server.cert`
- `--tls-private-key-file=/srv/kubernetes/server.key`
- `--admission-control=$RECOMMENDED_LIST`
  - See admission controllers for recommended arguments.
- `--allow-privileged=true`, only if you trust your cluster user to run pods as root.

If you are following the firewall-only security approach, then use these arguments:

- `--token-auth-file=/dev/null`
- `--insecure-bind-address=$MASTER_IP`
- `--advertise-address=$MASTER_IP`

If you are using the HTTPS approach, then set:

- `--client-ca-file=/srv/kubernetes/ca.crt`
- `--token-auth-file=/srv/kubernetes/known_tokens.csv`
- `--basic-auth-file=/srv/kubernetes/basic_auth.csv`

This pod mounts several node file system directories using the `hostPath` volumes. Their purposes are:

- The `/etc/ssl` mount allows the apiserver to find the SSL root certs so it can authenticate external services, such as a cloud provider.
  - This is not required if you do not use a cloud provider (e.g. bare-metal).
- The `/srv/kubernetes` mount allows the apiserver to read certs and credentials stored on the node disk. These could instead be stored on a persistent disk, such as a GCE PD, or baked into the image.

- Optionally, you may want to mount `/var/log` as well and redirect output there (not shown in template).
  - Do this if you prefer your logs to be accessible from the root filesystem with tools like journalctl.

TODO document proxy-ssh setup.

Cloud Providers

Apiserver supports several cloud providers.

- options for `--cloud-provider` flag are `aws`, `azure`, `cloudstack`, `fake`, `gce`, `mesos`, `openstack`, `ovirt`, `photon`, `rackspace`, `vsphere`, or unset.
- unset used for e.g. bare metal setups.
- support for new IaaS is added by contributing code here

Some cloud providers require a config file. If so, you need to put config file into apiserver image or mount through hostPath.

- `--cloud-config=` set if cloud provider requires a config file.
- Used by `aws`, `gce`, `mesos`, `openshift`, `ovirt` and `rackspace`.
- You must put config file into apiserver image or mount through hostPath.
- Cloud config file syntax is Gcfg.
- AWS format defined by type AWSCloudConfig
- There is a similar type in the corresponding file for other cloud providers.
- GCE example: search for `gce.conf` in this file

Scheduler pod template

Complete this template for the scheduler pod:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-scheduler"
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-scheduler",
        "image": "$HYBERKUBE_IMAGE",
        "command": [
          "/hyperkube",
          "scheduler",
          "--master=127.0.0.1:8080",
          "$SCHEDULER_FLAG1",
          ...
          "$SCHEDULER_FLAGN"
        ],
        "livenessProbe": {
          "httpGet": {
            "scheme": "HTTP",
            "host": "127.0.0.1",
            "port": 10251,
            "path": "/healthz"
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ]
  }
}
```

Typically, no additional flags are required for the scheduler.

Optionally, you may want to mount `/var/log` as well and redirect output there.

Controller Manager Template

Template for controller manager pod:

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
  },
  "spec": {
    "hostNetwork": true,
    "containers": [
      {
        "name": "kube-controller-manager",
        "image": "$HYPERKUBE_IMAGE",
        "command": [
          "/hyperkube",
          "controller-manager",
          "$CNTRLMNGR_FLAG1",
          ...
          "$CNTRLMNGR_FLAGN"
        ],
        "volumeMounts": [
          {
            "name": "srvkube",
            "mountPath": "/srv/kubernetes",
            "readOnly": true
          },
          {
            "name": "etcssl",
            "mountPath": "/etc/ssl",
            "readOnly": true
          }
        ],
        "livenessProbe": {
          "httpGet": {
            "scheme": "HTTP",
            "host": "127.0.0.1",
            "port": 10252,
            "path": "/healthz"
          },
          "initialDelaySeconds": 15,
          "timeoutSeconds": 15
        }
      }
    ],
    "volumes": [
      {
        "name": "srvkube",
        "hostPath": {
          "path": "/srv/kubernetes"
        }
      },
      {
        "name": "etcssl",
        "hostPath": {
          "path": "/etc/ssl"
        }
      }
      ]
    }
  }
```

Flags to consider using with controller manager:

- `--cluster-cidr=`, the CIDR range for pods in cluster.
- `--allocate-node-cidrs=`, if you are using `--cloud-provider=`, allocate and set the CIDRs for pods on the cloud provider.
- `--cloud-provider=` and `--cloud-config` as described in apiserver section.
- `--service-account-private-key-file=/srv/kubernetes/server.key`, used by the service account feature.
- `--master=127.0.0.1:8080`

Starting and Verifying Apiserver, Scheduler, and Controller Manager

Place each completed pod template into the kubelet config dir (whatever `--config=` argument of kubelet is set to, typically `/etc/kubernetes/manifests`). The order does not matter: scheduler and controller manager will retry reaching the apiserver until it is up.

Use `ps` or `docker ps` to verify that each process has started. For example, verify that kubelet has started a container for the apiserver like this:

```
$ sudo docker ps | grep apiserver:
5783290746d5        gcr.io/google_containers/kube-apiserver:e36bf367342b5a80d7467f
```

Then try to connect to the apiserver:

```
$ echo $(curl -s http://localhost:8080/healthz)
ok
$ curl -s http://localhost:8080/api
{
  "versions": [
    "v1"
  ]
}
```

If you have selected the `--register-node=true` option for kubelets, they will now begin self-registering with the apiserver. You should soon be able to see all your nodes by running the `kubectl get nodes` command. Otherwise, you will need to manually create node objects.

Starting Cluster Services

You will want to complete your Kubernetes clusters by adding cluster-wide services. These are sometimes called *addons*, and an overview of their purpose is in the admin guide.

Notes for setting up each cluster service are given below:

- Cluster DNS:
  - Required for many Kubernetes examples
  - Setup instructions
  - Admin Guide
- Cluster-level Logging
  - Cluster-level Logging Overview
  - Cluster-level Logging with Elasticsearch
  - Cluster-level Logging with Stackdriver Logging
- Container Resource Monitoring
  - Setup instructions
- GUI
  - Setup instructions cluster.

Troubleshooting

Running validate-cluster

`cluster/validate-cluster.sh` is used by `cluster/kube-up.sh` to determine if the cluster start succeeded.

Example usage and output:

```
KUBECTL_PATH=$(which kubectl) NUM_NODES=3 KUBERNETES_PROVIDER=local cluster/valida
Found 3 node(s).
NAME                    STATUS    AGE      VERSION
node1.local             Ready     1h       v1.6.9+a3d1dfa6f4335
node2.local             Ready     1h       v1.6.9+a3d1dfa6f4335
node3.local             Ready     1h       v1.6.9+a3d1dfa6f4335
Validate output:
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-1               Healthy   {"health": "true"}
etcd-2               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
Cluster validation succeeded
```

Inspect pods and services

Try to run through the "Inspect your cluster" section in one of the other Getting Started Guides, such as GCE. You should see some services. You should also see "mirror pods" for the apiserver, scheduler and controller-manager, plus any add-ons you started.

Try Examples

At this point you should be able to run through one of the basic examples, such as the nginx example.

Running the Conformance Test

You may want to try to run the Conformance test. Any failures may give a hint as to areas that need more attention.

Networking

The nodes must be able to connect to each other using their private IP. Verify this by pinging or SSH-ing from one node to another.

Getting Help

If you run into trouble, please see the section on troubleshooting, post to the kubernetes-users group, or come ask questions on Slack.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
any	any	any	any	docs		Community (@erictune)

For support level information on all solutions, see the Table of solutions chart.

Deprecated Alternatives

Stop. These guides are superseded by Minikube. They are only listed here for completeness.

- Using Vagrant
- *Advanced:* Directly using Kubernetes raw binaries (Linux Only)

Running Kubernetes on Google Compute Engine

The example below creates a Kubernetes cluster with 4 worker node Virtual Machines and a master Virtual Machine (i.e. 5 VMs in your cluster). This cluster is set up and controlled from your workstation (or wherever you find convenient).

Before you start

If you want a simplified getting started experience and GUI for managing clusters, please consider trying Google Container Engine (GKE) for hosted cluster installation and management.

If you want to use custom binaries or pure open source Kubernetes, please continue with the instructions below.

Prerequisites

1. You need a Google Cloud Platform account with billing enabled. Visit the Google Developers Console for more details.
2. Install `gcloud` as necessary. `gcloud` can be installed as a part of the Google Cloud SDK.
3. Enable the Compute Engine Instance Group Manager API in the Google Cloud developers console.
4. Make sure that gcloud is set to use the Google Cloud Platform project you want. You can check the current project using `gcloud config list project` and change it via `gcloud config set project <project-id>`.
5. Make sure you have credentials for GCloud by running `gcloud auth login`.
6. (Optional) In order to make API calls against GCE, you must also run `gcloud auth application-default login`.
7. Make sure you can start up a GCE VM from the command line. At least make sure you can do the Create an instance part of the GCE Quickstart.
8. Make sure you can SSH into the VM without interactive prompts. See the Log in to the instance part of the GCE Quickstart.

Starting a cluster

You can install a client and start a cluster with either one of these commands (we list both in case only one is installed on your machine):

```
curl -sS https://get.k8s.io | bash
```

or

```
wget -q -O - https://get.k8s.io | bash
```

Once this command completes, you will have a master VM and four worker VMs, running as a Kubernetes cluster.

By default, some containers will already be running on your cluster. Containers like `fluentd` provide logging, while `heapster` provides monitoring services.

The script run by the commands above creates a cluster with the name/prefix “kubernetes”. It defines one specific cluster config, so you can’t run it more than once.

Alternately, you can download and install the latest Kubernetes release from this page, then run the `<kubernetes>/cluster/kube-up.sh` script to start the cluster:

```
cd kubernetes
cluster/kube-up.sh
```

If you want more than one cluster running in your project, want to use a different name, or want a different number of worker nodes, see the `<kubernetes>/cluster/gce/config-default.sh` file for more fine-grained configuration before you start up your cluster.

If you run into trouble, please see the section on troubleshooting, post to the kubernetes-users group, or come ask questions on Slack.

The next few steps will show you:

1. How to set up the command line client on your workstation to manage the cluster
2. Examples of how to use the cluster
3. How to delete the cluster
4. How to start clusters with non-default options (like larger clusters)

Installing the Kubernetes command line tools on your workstation

The cluster startup script will leave you with a running cluster and a `kubernetes` directory on your workstation.

The kubectl tool controls the Kubernetes cluster manager. It lets you inspect your cluster resources, create, delete, and update components, and much more. You will use it to look at your new cluster and bring up example apps.

You can use `gcloud` to install the `kubectl` command-line tool on your workstation:

```
gcloud components install kubectl
```

Note: The kubectl version bundled with `gcloud` may be older than the one downloaded by the get.k8s.io install script. See Installing kubectl document to see how you can set up the latest `kubectl` on your workstation.

Getting started with your cluster

Inspect your cluster

Once `kubectl` is in your path, you can use it to look at your cluster. E.g., running:

```
$ kubectl get --all-namespaces services
```

should show a set of services that look something like this:

```
NAMESPACE     NAME                  CLUSTER_IP       EXTERNAL_IP       PORT(S)
default       kubernetes            10.0.0.1         <none>            443/TCP
kube-system   kube-dns              10.0.0.2         <none>            53/TCP,53/U
kube-system   kube-ui               10.0.0.3         <none>            80/TCP
...
```

Similarly, you can take a look at the set of pods that were created during cluster startup. You can do this via the

```
$ kubectl get --all-namespaces pods
```

command.

You'll see a list of pods that looks something like this (the name specifics will be different):

```
NAMESPACE     NAME                                            READY     STATUS    R
kube-system   fluentd-cloud-logging-kubernetes-minion-63uo   1/1       Running   0
kube-system   fluentd-cloud-logging-kubernetes-minion-c1n9   1/1       Running   0
kube-system   fluentd-cloud-logging-kubernetes-minion-c4og   1/1       Running   0
kube-system   fluentd-cloud-logging-kubernetes-minion-ngua   1/1       Running   0
kube-system   kube-dns-v5-7ztia                               3/3       Running   0
kube-system   kube-ui-v1-curt1                                1/1       Running   0
kube-system   monitoring-heapster-v5-ex4u3                    1/1       Running   1
kube-system   monitoring-influx-grafana-v1-piled              2/2       Running   0
```

Some of the pods may take a few seconds to start up (during this time they'll show `Pending`), but check that they all show as `Running` after a short period.

Run some examples

Then, see a simple nginx example to try out your new cluster.

For more complete applications, please look in the examples directory. The guestbook example is a good "getting started" walkthrough.

Tearing down the cluster

To remove/delete/teardown the cluster, use the `kube-down.sh` script.

```
cd kubernetes
cluster/kube-down.sh
```

Likewise, the `kube-up.sh` in the same directory will bring it back up. You do not need to rerun the `curl` or `wget` command: everything needed to setup the Kubernetes cluster is now on your workstation.

Customizing

The script above relies on Google Storage to stage the Kubernetes release. It then will start (by default) a single master VM along with 4 worker VMs. You can tweak some of these parameters by editing `kubernetes/cluster/gce/config-default.sh` You can view a transcript of a successful cluster creation here.

Troubleshooting

Project settings

You need to have the Google Cloud Storage API, and the Google Cloud Storage JSON API enabled. It is activated by default for new projects. Otherwise, it can be done in the Google Cloud Console. See the Google Cloud Storage JSON API Overview for more details.

Also ensure that– as listed in the Prerequsites section– you've enabled the `Compute Engine Instance Group Manager API`, and can start up a GCE VM from the command line as in the GCE Quickstart instructions.

Cluster initialization hang

If the Kubernetes startup script hangs waiting for the API to be reachable, you can troubleshoot by SSHing into the master and node VMs and looking at logs such as `/var/log/startupscript.log`.

Once you fix the issue, you should run `kube-down.sh` **to cleanup** after the partial cluster creation, before running `kube-up.sh` to try again.

SSH

If you're having trouble SSHing into your instances, ensure the GCE firewall isn't blocking port 22 to your VMs. By default, this should work but if you have edited firewall rules or created a new non-default network, you'll need to expose it:

```
gcloud compute firewall-rules create default-ssh --network=<network-name> --description "SSH allowed from anywhere" --allow tcp:22
```

Additionally, your GCE SSH key must either have no passcode or you need to be using `ssh-agent`.

Networking

The instances must be able to connect to each other using their private IP. The script uses the "default" network which should have a firewall rule called "default-allow-internal" which allows traffic on any port on the private IPs. If this rule is missing from the default network or if you change the network being used in `cluster/config-default.sh` create a new rule with the following field values:

- Source Ranges: `10.0.0.0/8`
- Allowed Protocols and Port: `tcp:1-65535;udp:1-65535;icmp`

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
GCE	Saltstack	Debian	GCE	docs		Project

For support level information on all solutions, see the Table of solutions chart.

Further reading

Please see the Kubernetes docs for more details on administering and using a Kubernetes cluster.

Running Kubernetes on AWS EC2

Supported Production Grade Tools

- Kubernetes Operations - Production Grade K8s Installation, Upgrades, and Management. Supports running Debian, Ubuntu, CentOS, and RHEL in AWS.
- CoreOS Tectonic includes the open-source Tectonic Installer that creates Kubernetes clusters with Container Linux nodes on AWS.
- CoreOS originated and the Kubernetes Incubator maintains a CLI tool, `kube-aws`, that creates and manages Kubernetes clusters with Container Linux nodes, using AWS tools: EC2, CloudFormation and Autoscaling.

kube-up is no longer supported in kubernetes 1.6

`kube-up.sh` is a legacy tool for launching clusters. It is deprecated, and removed entirely from kubernetes 1.6.

Prerequisites

1. This is only supported for kubernetes 1.5 and earlier. Consider switching to one of the supported options.
2. You need an AWS account. Visit http://aws.amazon.com to get started
3. Install and configure the AWS Command Line Interface
4. We recommend installing using an account which has full access to the AWS APIs.

NOTE: This script uses the 'default' AWS profile by default. You may explicitly set the AWS profile to use using the `AWS_DEFAULT_PROFILE` environment variable:

```
export AWS_DEFAULT_PROFILE=myawsprofile
```

Cluster turnup

Supported procedure: get-kube

```
#Using wget
export KUBERNETES_PROVIDER=aws; wget -q -O - https://get.k8s.io | bash
#Using cURL
export KUBERNETES_PROVIDER=aws; curl -sS https://get.k8s.io | bash
```

NOTE: This script calls cluster/kube-up.sh which in turn calls cluster/aws/util.sh using cluster/aws/config-default.sh.

This process takes about 5 to 10 minutes. Once the cluster is up, the IP addresses of your master and node(s) will be printed, as well as information about the default services running in the cluster (monitoring, logging, dns). User credentials and security tokens are written in `~/.kube/config`, they will be necessary to use the CLI or the HTTP Basic Auth.

By default, the script will provision a new VPC and a 4 node k8s cluster in us-west-2a (Oregon) with EC2 instances running on Debian. You can override the variables defined in config-default.sh to change this behavior as follows:

```
export KUBE_AWS_ZONE=eu-west-1c
export NUM_NODES=2
export MASTER_SIZE=m3.medium
export NODE_SIZE=m3.medium
export AWS_S3_REGION=eu-west-1
export AWS_S3_BUCKET=mycompany-kubernetes-artifacts
export KUBE_AWS_INSTANCE_PREFIX=k8s
...
```

If you don't specify master and minion sizes, the scripts will attempt to guess the correct size of the master and worker nodes based on `${NUM_NODES}`. In version 1.3 these default are:

- For the master, for clusters of less than 5 nodes it will use an `m3.medium`, for 6-10 nodes it will use an `m3.large`; for 11-100 nodes it will use an `m3.xlarge`.
- For worker nodes, for clusters less than 50 nodes it will use a `t2.micro`, for clusters between 50 and 150 nodes it will use a `t2.small` and for clusters with greater than 150 nodes it will use a `t2.medium`.

WARNING: beware that `t2` instances receive a limited number of CPU credits per hour and might not be suitable for clusters where the CPU is used consistently. As a rough estimation, consider 15 pods/node the absolute limit a `t2.large` instance can handle before it starts exhausting its CPU credits steadily, although this number depends heavily on the usage.

In prior versions of Kubernetes, we defaulted the master node to a t2-class instance, but found that this sometimes gave hard-to-diagnose problems when the master ran out of memory or CPU credits. If you are running a test cluster and want to save money, you can specify `export MASTER_SIZE=t2.micro` but if your master pauses do check the CPU credits in the AWS console.

For production usage, we recommend at least `export MASTER_SIZE=m3.medium` and `export NODE_SIZE=m3.medium`. And once you get above a handful of nodes, be aware that one m3.large instance has more storage than two m3.medium instances, for the same price.

We generally recommend the m3 instances over the m4 instances, because the m3 instances include local instance storage. Historically local instance storage has been more reliable than AWS EBS, and performance should be more consistent. The ephemeral nature of this storage is a match for ephemeral container workloads also!

If you use an m4 instance, or another instance type which does not have local instance storage, you may want to increase the `NODE_ROOT_DISK_SIZE` value, although the default value of 32 is probably sufficient for the smaller instance types in the m4 family.

The script will also try to create or reuse a keypair called "kubernetes", and IAM profiles called "kubernetes-master" and "kubernetes-minion". If these already exist, make sure you want them to be used here.

NOTE: If using an existing keypair named "kubernetes" then you must set the `AWS_SSH_KEY` key to point to your private key.

Getting started with your cluster

Command line administration tool: **kubectl**

The cluster startup script will leave you with a `kubernetes` directory on your workstation. Alternately, you can download the latest Kubernetes release from this page.

Next, add the appropriate binary folder to your `PATH` to access kubectl:

```
# OS X
export PATH=<path/to/kubernetes-directory>/platforms/darwin/amd64:$PATH

# Linux
export PATH=<path/to/kubernetes-directory>/platforms/linux/amd64:$PATH
```

An up-to-date documentation page for this tool is available here: kubectl manual

By default, `kubectl` will use the `kubeconfig` file generated during the cluster startup for authenticating against the API. For more information, please read kubeconfig files

Examples

See a simple nginx example to try out your new cluster.

The "Guestbook" application is another popular example to get started with Kubernetes: guestbook example

For more complete applications, please look in the examples directory

Scaling the cluster

Adding and removing nodes through `kubectl` is not supported. You can still scale the amount of nodes manually through adjustments of the 'Desired' and 'Max' properties within the Auto Scaling Group, which was created during the installation.

Tearing down the cluster

Make sure the environment variables you used to provision your cluster are still exported, then call the following script inside the `kubernetes` directory:

```
cluster/kube-down.sh
```

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
AWS	kops	Debian	k8s (VPC)	docs		Community (@justinsb)
AWS	CoreOS	CoreOS	flannel	docs		Community

For support level information on all solutions, see the Table of solutions chart.

Further reading

Please see the Kubernetes docs for more details on administering and using a Kubernetes cluster.

Running Kubernetes on Azure

Azure Container Service

The Azure Container Service offers simple deployments of one of three open source orchestrators: DC/OS, Swarm, and Kubernetes clusters.

For an example of deploying a Kubernetes cluster onto Azure via the Azure Container Service:

Microsoft Azure Container Service - Kubernetes Walkthrough

Custom Deployments: ACS-Engine

The core of the Azure Container Service is **open source** and available on GitHub for the community to use and contribute to: **ACS-Engine**.

ACS-Engine is a good choice if you need to make customizations to the deployment beyond what the Azure Container Service officially supports. These customizations include deploying into existing virtual networks, utilizing multiple agent pools, and more. Some community contributions to ACS-Engine may even become features of the Azure Container Service.

The input to ACS-Engine is similar to the ARM template syntax used to deploy a cluster directly with the Azure Container Service. The resulting output is an Azure Resource Manager Template that can then be checked into source control and can then be used to deploy Kubernetes clusters into Azure.

You can get started quickly by following the **ACS-Engine Kubernetes Walkthrough**.

CoreOS Tectonic for Azure

The CoreOS Tectonic Installer for Azure is **open source** and available on GitHub for the community to use and contribute to: **Tectonic Installer**.

Tectonic Installer is a good choice when you need to make cluster customizations as it is built on Hashicorp's Terraform Azure Resource Manager (ARM) provider. This enables users to customize or integrate using familiar Terraform tooling.

You can get started using the Tectonic Installer for Azure Guide.

Running Kubernetes on Alibaba Cloud

Alibaba Cloud Container Service

The Alibaba Cloud Container Service lets you run and manage Docker applications on a cluster of Alibaba Cloud ECS instances. It supports the popular open source container orchestrators: Docker Swarm and Kubernetes.

To simplify cluster deployment and management, use Kubernetes Suppport for Alibaba Cloud Container Service. You can get started quickly by following the Kubernetes walk-through, and there are some tutorials for Kubernetes Support on Alibaba Cloud in Chinese.

To use custom binaries or open source Kubernetes, follow the instructions below.

Custom Deployments

The source code for Kubernetes with Alibaba Cloud provider implmenetation is open source and available on GitHub.

For more information, see “Quick deployment of Kubernetes - VPC environment on Alibaba Cloud” in English and Chinese.

Running Kubernetes on CenturyLink Cloud

These scripts handle the creation, deletion and expansion of Kubernetes clusters on CenturyLink Cloud.

You can accomplish all these tasks with a single command. We have made the Ansible playbooks used to perform these tasks available here.

Find Help

If you run into any problems or want help with anything, we are here to help. Reach out to use via any of the following ways: - Submit a github issue - Send an email to Kubernetes AT ctl DOT io - Visit http://info.ctl.io/kubernetes

Clusters of VMs or Physical Servers, your choice.

- We support Kubernetes clusters on both Virtual Machines or Physical Servers. If you want to use physical servers for the worker nodes (minions), simple use the –minion_type=bareMetal flag.
- For more information on physical servers, visit: https://www.ctl.io/bare-metal/)
- Physical serves are only available in the VA1 and GB3 data centers.
- VMs are available in all 13 of our public cloud locations

Requirements

The requirements to run this script are: - A linux administrative host (tested on ubuntu and OSX) - python 2 (tested on 2.7.11) - pip (installed with python as of 2.7.9) - git - A CenturyLink Cloud account with rights to create new hosts - An active VPN connection to the CenturyLink Cloud from your linux host

Script Installation

After you have all the requirements met, please follow these instructions to install this script.

1) Clone this repository and cd into it.

```
git clone https://github.com/CenturyLinkCloud/adm-kubernetes-on-clc
```

2) Install all requirements, including * Ansible * CenturyLink Cloud SDK * Ansible Modules

```
sudo pip install -r ansible/requirements.txt
```

3) Create the credentials file from the template and use it to set your ENV variables

```
cp ansible/credentials.sh.template ansible/credentials.sh
vi ansible/credentials.sh
source ansible/credentials.sh
```

4) Grant your machine access to the CenturyLink Cloud network by using a VM inside the network or configuring a VPN connection to the CenturyLink Cloud network.

Script Installation Example: Ubuntu 14 Walkthrough

If you use an ubuntu 14, for your convenience we have provided a step by step guide to install the requirements and install the script.

```
 # system
apt-get update
apt-get install -y git python python-crypto
curl -O https://bootstrap.pypa.io/get-pip.py
python get-pip.py

 # installing this repository
mkdir -p ~home/k8s-on-clc
cd ~home/k8s-on-clc
git clone https://github.com/CenturyLinkCloud/adm-kubernetes-on-clc.git
cd adm-kubernetes-on-clc/
pip install -r requirements.txt

 # getting started
cd ansible
cp credentials.sh.template credentials.sh; vi credentials.sh
source credentials.sh
```

Cluster Creation

To create a new Kubernetes cluster, simply run the kube-up.sh script. A complete list of script options and some examples are listed below.

```
CLC_CLUSTER_NAME=[name of kubernetes cluster]
cd ./adm-kubernetes-on-clc
bash kube-up.sh -c="$CLC_CLUSTER_NAME"
```

It takes about 15 minutes to create the cluster. Once the script completes, it will output some commands that will help you setup kubectl on your machine to point to the new cluster.

When the cluster creation is complete, the configuration files for it are stored locally on your administrative host, in the following directory

```
> CLC_CLUSTER_HOME=$HOME/.clc_kube/$CLC_CLUSTER_NAME/
```

Cluster Creation: Script Options

```
Usage: kube-up.sh [OPTIONS]
Create servers in the CenturyLinkCloud environment and initialize a Kubernetes clu
Environment variables CLC_V2_API_USERNAME and CLC_V2_API_PASSWD must be set in
order to access the CenturyLinkCloud API

All options (both short and long form) require arguments, and must include "="
between option name and option value.

     -h (--help)                   display this help and exit
     -c= (--clc_cluster_name=)     set the name of the cluster, as used in CLC gro
     -t= (--minion_type=)          standard -> VM (default), bareMetal -> physical
     -d= (--datacenter=)           VA1 (default)
     -m= (--minion_count=)         number of kubernetes minion nodes
     -mem= (--vm_memory=)          number of GB ram for each minion
     -cpu= (--vm_cpu=)             number of virtual cps for each minion node
     -phyid= (--server_conf_id=)   physical server configuration id, one of
                                      physical_server_20_core_conf_id
                                      physical_server_12_core_conf_id
                                      physical_server_4_core_conf_id (default)
     -etcd_separate_cluster=yes    create a separate cluster of three etcd nodes,
                                   otherwise run etcd on the master node
```

Cluster Expansion

To expand an existing Kubernetes cluster, run the `add-kube-node.sh` script. A complete list of script options and some examples are listed [below]. This script must be run from the same host that created the cluster (or a host that has the cluster artifact files stored in `~/.clc_kube/$cluster_name`).

```
cd ./adm-kubernetes-on-clc
bash add-kube-node.sh -c="name_of_kubernetes_cluster" -m=2
```

Cluster Expansion: Script Options

```
Usage: add-kube-node.sh [OPTIONS]
Create servers in the CenturyLinkCloud environment and add to an
existing CLC kubernetes cluster

Environment variables CLC_V2_API_USERNAME and CLC_V2_API_PASSWD must be set in
order to access the CenturyLinkCloud API

     -h (--help)                   display this help and exit
     -c= (--clc_cluster_name=)     set the name of the cluster, as used in CLC gro
     -m= (--minion_count=)         number of kubernetes minion nodes to add
```

Cluster Deletion

There are two ways to delete an existing cluster:

1) Use our python script:

```
python delete_cluster.py --cluster=clc_cluster_name --datacenter=DC1
```

2) Use the CenturyLink Cloud UI. To delete a cluster, log into the CenturyLink Cloud control portal and delete the parent server group that contains the Kubernetes Cluster. We hope to add a scripted option to do this soon.

Examples

Create a cluster with name of k8s_1, 1 master node and 3 worker minions (on physical machines), in VA1

```
bash kube-up.sh --clc_cluster_name=k8s_1 --minion_type=bareMetal --minion_count=3
```

Create a cluster with name of k8s_2, an ha etcd cluster on 3 VMs and 6 worker minions (on VMs), in VA1

```
bash kube-up.sh --clc_cluster_name=k8s_2 --minion_type=standard --minion_count=6 -
```

Create a cluster with name of k8s_3, 1 master node, and 10 worker minions (on VMs) with higher mem/cpu, in UC1:

```
bash kube-up.sh --clc_cluster_name=k8s_3 --minion_type=standard --minion_count=10
```

Cluster Features and Architecture

We configure the Kubernetes cluster with the following features:

- KubeDNS: DNS resolution and service discovery
- Heapster/InfluxDB: For metric collection. Needed for Grafana and auto-scaling.
- Grafana: Kubernetes/Docker metric dashboard
- KubeUI: Simple web interface to view Kubernetes state
- Kube Dashboard: New web interface to interact with your cluster

We use the following to create the Kubernetes cluster:

- Kubernetes 1.1.7
- Ubuntu 14.04
- Flannel 0.5.4
- Docker 1.9.1-0~trusty
- Etcd 2.2.2

Optional add-ons

- Logging: We offer an integrated centralized logging ELK platform so that all Kubernetes and docker logs get sent to the ELK stack. To install the ELK stack and configure Kubernetes to send logs to it, follow the log aggregation documentation. Note: We don't install this by default as the footprint isn't trivial.

Cluster management

The most widely used tool for managing a Kubernetes cluster is the command-line utility `kubectl`. If you do not already have a copy of this binary on your administrative machine, you may run the script `install_kubectl.sh` which will download it and install it in `/usr/bin/local`.

The script requires that the environment variable `CLC_CLUSTER_NAME` be defined

`install_kubectl.sh` also writes a configuration file which will embed the necessary authentication certificates for the particular cluster. The configuration file is written to the `${CLC_CLUSTER_HOME}/kube` directory

```
export KUBECONFIG=${CLC_CLUSTER_HOME}/kube/config
kubectl version
kubectl cluster-info
```

Accessing the cluster programmatically

It's possible to use the locally stored client certificates to access the apiserver. For example, you may want to use any of the Kubernetes API client libraries to program against your Kubernetes cluster in the programming language of your choice.

To demonstrate how to use these locally stored certificates, we provide the following example of using `curl` to communicate to the master apiserver via https:

```
curl \
   --cacert ${CLC_CLUSTER_HOME}/pki/ca.crt  \
   --key ${CLC_CLUSTER_HOME}/pki/kubecfg.key \
   --cert ${CLC_CLUSTER_HOME}/pki/kubecfg.crt  https://${MASTER_IP}:6443
```

But please note, this *does not* work out of the box with the `curl` binary distributed with OSX.

Accessing the cluster with a browser

We install two UIs on Kubernetes. The original KubeUI and the newer kube dashboard. When you create a cluster, the script should output URLs for these interfaces like this:

KubeUI is running at `https://${MASTER_IP}:6443/api/v1/namespaces/kube-system/services/kube-ui/proxy` kubernetes-dashboard is running at `https://${MASTER_IP}:6443/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy`

Note on Authentication to the UIs: The cluster is set up to use basic authentication for the user *admin*. Hitting the url at `https://${MASTER_IP}:6443` will require accepting the self-signed certificate from the apiserver, and then presenting the admin password written to file at:

`> _${CLC_CLUSTER_HOME}/kube/admin_password.txt_`

Configuration files

Various configuration files are written into the home directory *CLC_CLUSTER_HOME* under `.clc_kube/${CLC_CLUSTER_NAME}` in several subdirectories. You can use these files to access the cluster from machines other than where you created the cluster from.

- `config/` : Ansible variable files containing parameters describing the master and minion hosts
- `hosts/` : hosts files listing access information for the ansible playbooks
- `kube/` : `kubectl` configuration files, and the basic-authentication password for admin access to the Kubernetes API
- `pki/` : public key infrastructure files enabling TLS communication in the cluster
- `ssh/` : SSH keys for root access to the hosts

kubectl usage examples

There are a great many features of *kubectl*. Here are a few examples

List existing nodes, pods, services and more, in all namespaces, or in just one:

```
kubectl get nodes
kubectl get --all-namespaces services
kubectl get --namespace=kube-system replicationcontrollers
```

The Kubernetes API server exposes services on web URLs, which are protected by requiring client certificates. If you run a kubectl proxy locally, `kubectl` will provide the necessary certificates and serve locally over http.

```
kubectl proxy -p 8001
```

Then, you can access urls like `http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kube-ui/proxy/` without the need for client certificates in your browser.

What Kubernetes features do not work on CenturyLink Cloud

These are the known items that don't work on CenturyLink cloud but do work on other cloud providers:

- At this time, there is no support services of the type LoadBalancer. We are actively working on this and hope to publish the changes sometime around April 2016.
- At this time, there is no support for persistent storage volumes provided by CenturyLink Cloud. However, customers can bring their own persistent storage offering. We ourselves use Gluster.

Ansible Files

If you want more information about our Ansible files, please read this file

Further reading

Please see the Kubernetes docs for more details on administering and using a Kubernetes cluster.

Running Kubernetes on Multiple Clouds with Stackpoint.io

Introduction

StackPointCloud is the universal control plane for Kubernetes Anywhere. StackPointCloud allows you to deploy and manage a Kubernetes cluster to the cloud provider of your choice in 3 steps using a web-based interface.

AWS

To create a Kubernetes cluster on AWS, you will need an Access Key ID and a Secret Access Key from AWS.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select Amazon Web Services (AWS).

Configure Your Provider

Add your Access Key ID and a Secret Access Key from AWS. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on AWS, consult the Kubernetes documentation.

GCE

To create a Kubernetes cluster on GCE, you will need the Service Account JSON Data from Google.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select Google Compute Engine (GCE).

Configure Your Provider

Add your Service Account JSON Data from Google. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on GCE, consult the Kubernetes documentation.

GKE

To create a Kubernetes cluster on GKE, you will need the Service Account JSON Data from Google.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select Google Container Engine (GKE).

Configure Your Provider

Add your Service Account JSON Data from Google. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on GKE, consult the official documentation.

DigitalOcean

To create a Kubernetes cluster on DigitalOcean, you will need a DigitalOcean API Token.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select DigitalOcean.

Configure Your Provider

Add your DigitalOcean API Token. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on DigitalOcean, consult the official documentation.

Microsoft Azure

To create a Kubernetes cluster on Microsoft Azure, you will need an Azure Subscription ID, Username/Email, and Password.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select Microsoft Azure.

Configure Your Provider

Add your Azure Subscription ID, Username/Email, and Password. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on Azure, consult the Kubernetes documentation.

Packet

To create a Kubernetes cluster on Packet, you will need a Packet API Key.

Choose a Provider

Log in to stackpoint.io with a GitHub, Google, or Twitter account.

Click **+ADD A CLUSTER NOW**.

Click to select Packet.

Configure Your Provider

Add your Packet API Key. Select your default StackPointCloud SSH keypair, or click **ADD SSH KEY** to add a new keypair.

Click **SUBMIT** to submit the authorization information.

Configure Your Cluster

Choose any extra options you may want to include with your cluster, then click **SUBMIT** to create the cluster.

Running the Cluster

You can monitor the status of your cluster and suspend or delete it from your stackpoint.io dashboard.

For information on using and managing a Kubernetes cluster on Packet, consult the official documentation.

Installing Kubernetes on AWS with kops

Overview

This quickstart shows you how to easily install a Kubernetes cluster on AWS. It uses a tool called `kops`.

kops is an opinionated provisioning system:

- Fully automated installation
- Uses DNS to identify clusters
- Self-healing: everything runs in Auto-Scaling Groups
- Limited OS support (Debian preferred, Ubuntu 16.04 supported, early support for CentOS & RHEL)
- High-Availability support
- Can directly provision, or generate terraform manifests

If your opinions differ from these you may prefer to build your own cluster using kubeadm as a building block. kops builds on the kubeadm work.

Creating a cluster

(1/5) Install kops

Requirements

You must have kubectl installed in order for kops to work.

Installation

Download kops from the releases page (it is also easy to build from source):

On MacOS:

```
wget https://github.com/kubernetes/kops/releases/download/1.7.0/kops-darwin-amd64
chmod +x kops-darwin-amd64
mv kops-darwin-amd64 /usr/local/bin/kops
# you can also install using Homebrew
brew update && brew install kops
```

On Linux:

```
wget https://github.com/kubernetes/kops/releases/download/1.7.0/kops-linux-amd64
chmod +x kops-linux-amd64
mv kops-linux-amd64 /usr/local/bin/kops
```

(2/5) Create a route53 domain for your cluster

kops uses DNS for discovery, both inside the cluster and so that you can reach the kubernetes API server from clients.

kops has a strong opinion on the cluster name: it should be a valid DNS name. By doing so you will no longer get your clusters confused, you can share clusters with your colleagues unambiguously, and you can reach them without relying on remembering an IP address.

You can, and probably should, use subdomains to divide your clusters. As our example we will use `useast1.dev.example.com`. The API server endpoint will then be `api.useast1.dev.example.com`.

A Route53 hosted zone can serve subdomains. Your hosted zone could be `useast1.dev.example.com`, but also `dev.example.com` or even `example.com`. kops works with any of these, so typically you choose for organization reasons (e.g. you are allowed to create records under `dev.example.com`, but not under `example.com`).

Let's assume you're using `dev.example.com` as your hosted zone. You create that hosted zone using the normal process, or with a command such as `aws route53 create-hosted-zone --name dev.example.com --caller-reference 1`.

You must then set up your NS records in the parent domain, so that records in the domain will resolve. Here, you would create NS records in `example.com` for `dev`. If it is a root domain name you would configure the NS records at your domain registrar (e.g. `example.com` would need to be configured where you bought `example.com`).

This step is easy to mess up (it is the #1 cause of problems!) You can double-check that your cluster is configured correctly if you have the dig tool by running:

```
dig NS dev.example.com
```

You should see the 4 NS records that Route53 assigned your hosted zone.

(3/5) Create an S3 bucket to store your clusters state

kops lets you manage your clusters even after installation. To do this, it must keep track of the clusters that you have created, along with their configuration, the keys they are using etc. This information is stored in an S3 bucket. S3 permissions are used to control access to the bucket.

Multiple clusters can use the same S3 bucket, and you can share an S3 bucket between your colleagues that administer the same clusters - this is much easier than passing around kubecfg files. But anyone with access to the S3 bucket will have administrative access to all your clusters, so you don't want to share it beyond the operations team.

So typically you have one S3 bucket for each ops team (and often the name will correspond to the name of the hosted zone above!)

In our example, we chose `dev.example.com` as our hosted zone, so let's pick `clusters.dev.example.com` as the S3 bucket name.

- Export `AWS_PROFILE` (if you need to select a profile for the AWS CLI to work)
- Create the S3 bucket using `aws s3 mb s3://clusters.dev.example.com`
- You can `export KOPS_STATE_STORE=s3://clusters.dev.example.com` and then kops will use this location by default. We suggest putting this in your bash profile or similar.

(4/5) Build your cluster configuration

Run "kops create cluster" to create your cluster configuration:

```
kops create cluster --zones=us-east-1c useast1.dev.example.com
```

kops will create the configuration for your cluster. Note that it *only* creates the configuration, it does not actually create the cloud resources - you'll do that in the next step with a `kops update cluster`. This give you an opportunity to review the configuration or change it.

It prints commands you can use to explore further:

- List your clusters with: `kops get cluster`
- Edit this cluster with: `kops edit cluster useast1.dev.example.com`
- Edit your node instance group: `kops edit ig --name=useast1.dev.example.com nodes`
- Edit your master instance group: `kops edit ig --name=useast1.dev.example.com master-us-east-1c`

If this is your first time using kops, do spend a few minutes to try those out! An instance group is a set of instances, which will be registered as kubernetes nodes. On AWS this is implemented via auto-scaling-groups. You can have several instance groups, for example if you wanted nodes that are a mix of spot and on-demand instances, or GPU and non-GPU instances.

(5/5) Create the cluster in AWS

Run "kops update cluster" to create your cluster in AWS:

`kops update cluster useast1.dev.example.com --yes`

That takes a few seconds to run, but then your cluster will likely take a few minutes to actually be ready. `kops update cluster` will be the tool you'll use whenever you change the configuration of your cluster; it applies the changes you have made to the configuration to your cluster - reconfiguring AWS or kubernetes as needed.

For example, after you `kops edit ig nodes`, then `kops update cluster --yes` to apply your configuration, and sometimes you will also have to `kops rolling-update cluster` to roll out the configuration immediately.

Without `--yes`, `kops update cluster` will show you a preview of what it is going to do. This is handy for production clusters!

Explore other add-ons

See the list of add-ons to explore other add-ons, including tools for logging, monitoring, network policy, visualization & control of your Kubernetes cluster.

What's next

- Learn more about Kubernetes concepts and `kubectl`.
- Learn about `kops` advanced usage

Cleanup

- To delete you cluster: `kops delete cluster useast1.dev.example.com --yes`

Feedback

- Slack Channel: #sig-aws has a lot of kops users
- GitHub Issues

Installing Kubernetes On-premises/Cloud Providers with Kubespray

Overview

This quickstart helps to install a Kubernetes cluster hosted on GCE, Azure, OpenStack, AWS, or Baremetal with Kubespray.

Kubespray is a composition of Ansible playbooks, inventory, provisioning tools, and domain knowledge for generic OS/Kubernetes clusters configuration management tasks. Kubespray provides:

- a highly available cluster
- composable attributes
- support for most popular Linux distributions
- continuous integration tests

To choose a tool which best fits your use case, read this comparison to kubeadm and kops.

Creating a cluster

(1/5) Meet the underlay requirements

Provision servers with the following requirements:

- `Ansible v2.3` (or newer)
- `Jinja 2.9` (or newer)
- `python-netaddr` installed on the machine that running Ansible commands
- Target servers must have access to the Internet in order to pull docker images

- Target servers are configured to allow IPv4 forwarding
- Target servers have SSH connectivity (tcp/22) directly to your nodes or through a bastion host/ssh jump box
- Target servers have a privileged user
- Your SSH key must be copied to all the servers that are part of your inventory
- Firewall rules configured properly to allow Ansible and Kubernetes components to communicate
- If using a cloud provider, you must have the appropriate credentials available and exported as environment variables

Kubespray provides the following utilities to help provision your environment:

- Terraform scripts for the following cloud providers:
  - AWS
  - OpenStack
- kubespray-cli

Note: kubespray-cli is no longer actively maintained. {. :note}

(2/5) Compose an inventory file

After you provision your servers, create an inventory file for Ansible. You can do this manually or via a dynamic inventory script. For more information, see "Building your own inventory".

(3/5) Plan your cluster deployment

Kubespray provides the ability to customize many aspects of the deployment:

- CNI (networking) plugins
- DNS configuration
- Choice of control plane: native/binary or containerized with docker or rkt)
- Component versions

- Calico route reflectors
- Component runtime options
- Certificate generation methods

Kubespray customizations can be made to a variable file. If you are just getting started with Kubespray, consider using the Kubespray defaults to deploy your cluster and explore Kubernetes.

(4/5) Deploy a Cluster

Next, deploy your cluster with one of two methods:

- ansible-playbook.
- kubespray-cli tool

> **Note:** kubespray-cli is no longer actively maintained.

Both methods run the default cluster definition file.

Large deployments (100+ nodes) may require specific adjustments for best results.

(5/5) Verify the deployment

Kubespray provides a way to verify inter-pod connectivity and DNS resolve with Netchecker. Netchecker ensures the netchecker-agents pods can resolve DNS requests and ping each over within the default namespace. Those pods mimic similar behavior of the rest of the workloads and serve as cluster health indicators.

Cluster operations

Kubespray provides additional playbooks to manage your cluster: *scale* and *upgrade*.

Scale your cluster

You can scale your cluster by running the scale playbook. For more information, see “Adding nodes”.

Upgrade your cluster

You can upgrade your cluster by running the upgrade-cluster playbook. For more information, see “Upgrades”.

What’s next

Check out planned work on Kubespray’s roadmap.

Cleanup

You can reset your nodes and wipe out all components installed with Kubespray via the reset playbook.

Caution: When running the reset playbook, be sure not to accidentally target your production cluster!

Feedback

- Slack Channel: #kubespray
- GitHub Issues

Cloudstack

CloudStack is a software to build public and private clouds based on hardware virtualization principles (traditional IaaS). To deploy Kubernetes on CloudStack there are several possibilities depending on the Cloud being used and what images are made available. CloudStack also has a vagrant plugin available, hence Vagrant could be used to deploy Kubernetes either using the existing shell provisioner or using new Salt based recipes.

CoreOS templates for CloudStack are built nightly. CloudStack operators need to register this template in their cloud before proceeding with these Kubernetes deployment instructions.

This guide uses an Ansible playbook. This is completely automated, a single playbook deploys Kubernetes.

This Ansible playbook deploys Kubernetes on a CloudStack based Cloud using CoreOS images. The playbook, creates an ssh key pair, creates a security group and associated rules and finally starts coreOS instances configured via cloud-init.

Prerequisites

```
$ sudo apt-get install -y python-pip libssl-dev
$ sudo pip install cs
$ sudo pip install sshpubkeys
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

On CloudStack server you also have to install libselinux-python :

```
yum install libselinux-python
```

cs is a python module for the CloudStack API.

Set your CloudStack endpoint, API keys and HTTP method used.

You can define them as environment variables: `CLOUDSTACK_ENDPOINT`, `CLOUDSTACK_KEY`, `CLOUDSTACK_SECRET` and `CLOUDSTACK_METHOD`.

Or create a `~/.cloudstack.ini` file:

```
[cloudstack]
endpoint = <your cloudstack api endpoint>
key = <your api access key>
secret = <your api secret key>
method = post
```

We need to use the http POST method to pass the *large* userdata to the coreOS instances.

Clone the playbook

```
$ git clone https://github.com/apachecloudstack/k8s
$ cd kubernetes-cloudstack
```

Create a Kubernetes cluster

You simply need to run the playbook.

```
$ ansible-playbook k8s.yml
```

Some variables can be edited in the `k8s.yml` file.

```
vars:
  ssh_key: k8s
  k8s_num_nodes: 2
  k8s_security_group_name: k8s
  k8s_node_prefix: k8s2
  k8s_template: <templatename>
  k8s_instance_type: <serviceofferingname>
```

This will start a Kubernetes master node and a number of compute nodes (by default 2). The `instance_type` and `template` are specific, edit them to specify your CloudStack cloud specific template and instance type (i.e. service offering).

Check the tasks and templates in `roles/k8s` if you want to modify anything.

Once the playbook as finished, it will print out the IP of the Kubernetes master:

```
TASK: [k8s | debug msg='k8s master IP is {{ k8s_master.default_ip }}'] ********
```

SSH to it using the key that was created and using the *core* user and you can list the machines in your cluster:

```
$ ssh -i ~/.ssh/id_rsa_k8s core@<master IP>
$ fleetctl list-machines
MACHINE		IP		   METADATA
a017c422...	<node #1 IP>	role=node
ad13bf84...	<master IP>	   role=master
e9af8293...	<node #2 IP>	role=node
```

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
CloudStack	Ansible	CoreOS	flannel	docs		Community (@Guiques)

For support level information on all solutions, see the Table of solutions chart.

VMware vSphere

This page covers how to get started with deploying Kubernetes on vSphere and details for how to configure the vSphere Cloud Provider.

Getting started with the vSphere Cloud Provider

Kubernetes comes with *vSphere Cloud Provider*, a cloud provider for vSphere that allows Kubernetes Pods to use enterprise grade vSphere Storage.

Deploy Kubernetes on vSphere

To deploy Kubernetes on vSphere and use the vSphere Cloud Provider, see Kubernetes-Anywhere.

Detailed steps can be found at the getting started with Kubernetes-Anywhere on vSphere page.

vSphere Cloud Provider

vSphere Cloud Provider allows Kubernetes to use vSphere managed enterprise grade storage. It supports:

- Enterprise class services such as de-duplication and encryption with vSAN, QoS, high availability and data reliability.
- Policy based management at granularity of container volumes.
- Volumes, Persistent Volumes, Storage Classes, dynamic provisioning of volumes, and scalable deployment of Stateful Apps with StatefulSets.

For more detail visit vSphere Storage for Kubernetes Documentation.

Documentation for how to use vSphere managed storage can be found in the persistent volumes user guide and the volumes user guide.

Examples can be found here.

Enable vSphere Cloud Provider

If a Kubernetes cluster has not been deployed using Kubernetes-Anywhere, follow the instructions below to enable the vSphere Cloud Provider. These steps are not needed when using Kubernetes-Anywhere, they will be done as part of the deployment.

Step-1 Create a VM folder and move Kubernetes Node VMs to this folder.

Step-2 Make sure Node VM names must comply with the regex `[a-z](([-0-9a-z]+)?[0-9a-z])?(\.[a-z0-9](([-0-9a-z]+)?[0-9a-z])?)*`. If Node VMs do not comply with this regex, rename them and make it compliant to this regex.

Node VM names constraints:

- VM names can not begin with numbers.
- VM names can not have capital letters, any special characters except `.` and `-`.
- VM names can not be shorter than 3 chars and longer than 63.

Step-3 Enable disk UUID on Node virtual machines.

The disk.EnableUUID parameter must be set to “TRUE” for each Node VM. This step is necessary so that the VMDK always presents a consistent UUID to the VM, thus allowing the disk to be mounted properly.

For each of the virtual machine nodes that will be participating in the cluster, follow the steps below using GOVC tool

- Set up GOVC environment

```
export GOVC_URL='vCenter IP OR FQDN'
export GOVC_USERNAME='vCenter User'
export GOVC_PASSWORD='vCenter Password'
export GOVC_INSECURE=1
```

- Find Node VM Paths

```
govc ls /datacenter/vm/<vm-folder-name>
```

- Set disk.EnableUUID to true for all VMs

```
govc vm.change -e="disk.enableUUID=1" -vm='VM Path'
```

Note: If Kubernetes Node VMs are created from template VM then `disk.EnableUUID=1` can be set on the template VM. VMs cloned from this template, will automatically inherit this property.

Step-4 Create and assign Roles to the vSphere Cloud Provider user and vSphere entities.

Note: if you want to use Administrator account then this step can be skipped.

vSphere Cloud Provider requires the following minimal set of privileges to interact with vCenter. Please refer vSphere Documentation Center to know about steps for creating a Custom Role, User and Role Assignment.

Roles	Privileges	Entities	Propagate to Children
manage-k8s-node-vms	Resource.AssignVMToPool System.Anonymous System.Read System.View VirtualMachine.Config.AddExistingDisk VirtualMachine.Config.AddNewDisk VirtualMachine.Config.AddRemoveDevice VirtualMachine.Config.RemoveDisk VirtualMachine.Inventory.Create VirtualMachine.Inventory.Delete	Cluster, Hosts, VM Folder	Yes
manage-k8s-volumes	Datastore.AllocateSpace Datastore.FileManagement System.Anonymous System.Read System.View	Datastore	No
k8s-system-read-and-spbm-profile-view	StorageProfile.View System.Anonymous System.Read System.View	vCenter	No
ReadOnly	System.Anonymous System.Read System.View	Datacenter, Datastore Cluster, Datastore Storage Folder	No

Step-5 Create the vSphere cloud config file (`vsphere.conf`). Cloud config template can be found here.

This config file needs to be placed in the shared directory which should be accessible from kubelet container, controller-manager pod, and API server pod.

`vsphere.conf` **for Master Node:**

```
[Global]
        user = "vCenter username for cloud provider"
        password = "password"
        server = "IP/FQDN for vCenter"
        port = "443" #Optional
        insecure-flag = "1" #set to 1 if the vCenter uses a self-signed cert
        datacenter = "Datacenter name"
        datastore = "Datastore name" #Datastore to use for provisioning volumes us
        working-dir = "vCenter VM folder path in which node VMs are located"
        vm-name = "VM name of the Master Node" #Optional
        vm-uuid = "UUID of the Node VM" # Optional
[Disk]
    scsicontrollertype = pvscsi
```

Note: `vm-name` **parameter is introduced in 1.6.4 release.** Both `vm-uuid` and `vm-name` are optional parameters. If `vm-name` is specified then `vm-uuid` is not used. If both are not specified then kubelet will get vm-uuid from `/sys/class/dmi/id/product_serial` and query vCenter to find the Node VM's name.

`vsphere.conf` **for Worker Nodes:** (Only Applicable to 1.6.4 release and above. For older releases this file should have all the parameters specified in Master node's `vSphere.conf` file).

```
[Global]
        vm-name = "VM name of the Worker Node"
```

Below is summary of supported parameters in the `vsphere.conf` file

- `user` is the vCenter username for vSphere Cloud Provider.
- `password` is the password for vCenter user specified with `user` .
- `server` is the vCenter Server IP or FQDN

- `port` is the vCenter Server Port. Default is 443 if not specified.
- `insecure-flag` is set to 1 if vCenter used a self-signed certificate.
- `datacenter` is the name of the datacenter on which Node VMs are deployed.
- `datastore` is the default datastore to use for provisioning volumes using storage classes/dynamic provisioning.
- `vm-name` is recently added configuration parameter. This is optional parameter. When this parameter is present, `vsphere.conf` file on the worker node does not need vCenter credentials.

  Note: `vm-name` is added in the release 1.6.4. Prior releases does not support this parameter.
- `working-dir` can be set to empty (working-dir = “”), if Node VMs are located in the root VM folder.
- `vm-uuid` is the VM Instance UUID of virtual machine. `vm-uuid` can be set to empty (`vm-uuid = ""`). If set to empty, this will be retrieved from /sys/class/dmi/id/product_serial file on virtual machine (requires root access).
  - `vm-uuid` needs to be set in this format - `423D7ADC-F7A9-F629-8454-CE9615C810F1`
  - `vm-uuid` can be retrieved from Node Virtual machines using following command. This will be different on each node VM.

    ```
    cat /sys/class/dmi/id/product_serial | sed -e 's/^VMware-//' -e 's/-/ /' |
    ```
- `datastore` is the default datastore used for provisioning volumes using storage classes. If datastore is located in storage folder or datastore is member of datastore cluster, make sure to specify full datastore path. Make sure vSphere Cloud Provider user has Read Privilege set on the datastore cluster or storage folder to be able to find datastore.
  - For datastore located in the datastore cluster, specify datastore as mentioned below

    ```
    datastore = "DatastoreCluster/datastore1"
    ```

- For datastore located in the storage folder, specify datastore as mentioned below

```
datastore = "DatastoreStorageFolder/datastore1"
```

Step-6 Add flags to controller-manager, API server and Kubelet to enable vSphere Cloud Provider. * Add following flags to kubelet running on every node and to the controller-manager and API server pods manifest files.

```
--cloud-provider=vsphere
--cloud-config=<Path of the vsphere.conf file>
```

Manifest files for API server and controller-manager are generally located at `/etc/kubernetes/manifests`.

Step-7 Restart Kubelet on all nodes.

- Reload kubelet systemd unit file using `systemctl daemon-reload`
- Restart kubelet service using `systemctl restart kubelet.service`

Note: After enabling the vSphere Cloud Provider, Node names will be set to the VM names from the vCenter Inventory.

Known issues

Please visit known issues for the list of major known issues with Kubernetes vSphere Cloud Provider.

Support Level

For quick support please join VMware Code Slack (kubernetes) and post your question.

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Vmware vSphere	Kube-anywhere	Photon OS	Flannel	docs		Community (@abrarshivani), (@kerneltime), (@BaluDontu), (@luomiao), (@divyenpatel)

If you identify any issues/problems using the vSphere cloud provider, you can create an issue in our repo - VMware Kubernetes.

For support level information on all solutions, see the Table of solutions chart.

VMware Photon Controller

The example below creates a Kubernetes cluster using VMware's Photon Controller. The cluster will have one Kubernetes master and three Kubernetes nodes.

Prerequisites

1. You need administrator access to a VMware Photon Controller deployment. (Administrator access is only required for the initial setup: the actual creation of the cluster can be done by anyone.)

2. The Photon Controller CLI needs to be installed on the machine on which you'll be running kube-up. If you have go installed, this can be easily installed with:

```
go get github.com/vmware/photon-controller-cli/photon
```

3. `mkisofs` needs to be installed. The installation process creates a CD-ROM ISO image to bootstrap the VMs with cloud-init. If you are on a Mac, you can install this with brew:

```
brew install cdrtools
```

4. Several common tools need to be installed: `ssh` , `scp` , `openssl`
5. You should have an ssh public key installed. This will be used to give you access to the VM's user account, `kube` .
6. Get or build a binary release

Download VM Image

Download a prebuilt Debian 8.2 VMDK that we'll use as a base image:

```
curl --remote-name-all https://s3.amazonaws.com/photon-platform/artifacts/OS/debia
```

This is a base Debian 8.2 image with the addition of:

- openssh-server
- open-vm-tools
- cloud-init

Configure Photon Controller:

In order to deploy Kubernetes, you need to configure Photon Controller with:

- A tenant, with associated resource ticket
- A project within that tenant
- VM and disk flavors, to describe the VM characteristics
- An image: we'll use the one above

When you do this, you'll need to configure the `cluster/photon-controller/config-common.sh` file with the names of the tenant, project, flavors, and image.

If you prefer, you can use the provided `cluster/photon-controller/setup-prereq.sh` script to create these. Assuming the IP address of your Photon Controller is 192.0.2.2 (change as appropriate) and the downloaded image is kube.vmdk, you can run:

```
photon target set https://192.0.2.2
photon target login ...credentials...
cluster/photon-controller/setup-prereq.sh https://192.0.2.2 kube.vmdk
```

The `setup-prereq.sh` script will create the tenant, project, flavors, and image based on the same configuration file used by kube-up: `cluster/photon-controller/config-common.sh`. Note that it will create a resource ticket which limits how many VMs a tenant can create. You will want to change the resource ticket configuration in `config-common.sh` based on your actual Photon Controller deployment.

Configure kube-up

There are two files used to configure kube-up's interaction with Photon Controller:

1. `cluster/photon-controller/config-common.sh` has the most common parameters, including the names of the tenant, project, and image.
2. `cluster/photon-controller/config-default.sh` has more advanced parameters including the IP subnets to use, the number of nodes to create and which Kubernetes components to configure.

Both files have documentation to explain the different parameters.

Creating your Kubernetes cluster

To create your Kubernetes cluster we will run the standard `kube-up` command. As described above, the parameters that control kube-up's interaction with Photon Controller are specified in files, not on the command-line.

The time to deploy varies based on the number of nodes you create as well as the specifications of your Photon Controller hosts and network. Times vary from 10 - 30 minutes for a ten node cluster.

```
KUBERNETES_PROVIDER=photon-controller cluster/kube-up.sh
```

Once you have successfully reached this point, your Kubernetes cluster works just like any other.

Note that kube-up created a Kubernetes configuration file for you in `~/.kube/config`. This file will allow you to use the `kubectl` command. It contains the IP address of the Kubernetes master as well as the password for the `admin` user. If you wish to use the Kubernetes web-based user interface you will need this password. In the config file you'll see a section that look like the following: you use the password there. (Note that the output has been trimmed: the certificate data is much lengthier)

```
- name: photon-kubernetes
  user:
    client-certificate-data: Q2Vyd...
    client-key-data: LS0tL...
    password: PASSWORD-HERE
    username: admin
```

Removing your Kubernetes cluster

The recommended way to remove your Kubernetes cluster is with the `kube-down` command:

```
KUBERNETES_PROVIDER=photon-controller cluster/kube-down.sh
```

Your Kubernetes cluster is just a set of VMs: you can manually remove them if you need to.

Making services publicly accessible

There are multiple ways to make services publicly accessible in Kubernetes. Currently, the photon-controller support does not yet include built-in support for the LoadBalancer option.

Option 1: NodePort

One option is to use the NodePort option with a manually deployed balancer. Specifically:

Configure your service with the NodePort option. For example, this service uses the NodePort option. All Kubernetes nodes will listen on a port and forward network traffic to any pods in the service. In this case, Kubernetes will choose a random port, but it will be the same port on all nodes.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-demo-service
  labels:
    app: nginx-demo
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: nginx-demo
```

Next, create a new standalone VM (or VMs, for high availability) to act as a load balancer. For example, if you use haproxy, you could make a configuration similar to the one below. Note that this example assumes there are three Kubernetes nodes: you would adjust the configuration to reflect the actual nodes you have. Also note that port 30144 should be replaced with whatever NodePort was assigned by Kubernetes.

```
frontend nginx-demo
    bind *:30144
    mode http
    default_backend nodes
backend nodes
    mode http
    balance roundrobin
    option forwardfor
    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    server web0 192.0.2.2:30144 check
    server web1 192.0.2.3:30144 check
    server web2 192.0.2.4:30144 check
```

Option 2: Ingress Controller

Using an ingress controller may also be an appropriate solution. Note that it in a production environment it will also require an external load balancer. However, it may be simpler to manage because it will not require you to manually update the load balancer configuration, as above.

Details

Logging into VMs

When the VMs are created, a `kube` user is created (using cloud-init). The password for the kube user is the same as the administrator password for your Kubernetes master and can be found in your Kubernetes configuration file: see above to find it. The kube user will also authorize your ssh public key to log in. This is used during installation to avoid the need for passwords.

The VMs do have a root user, but ssh to the root user is disabled.

Networking

The Kubernetes cluster uses `kube-proxy` to configure the overlay network with iptables. Currently we do not support other overlay networks such as Weave or Calico.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Vmware Photon	Saltstack	Debian	OVS	docs		Community (@alainroy)

Kubernetes on DCOS

Mesosphere provides an easy option to provision Kubernetes onto DC/OS, offering:

- Pure upstream Kubernetes
- Single-click cluster provisioning
- Highly available and secure by default
- Kubernetes running alongside fast-data platforms (e.g. Akka, Cassandra, Kafka, Spark)

Official Mesosphere Guide

The canonical source of getting started on DC/OS is located in the quickstart repo.

CoreOS on libvirt

Highlights

- Super-fast cluster boot-up (few seconds instead of several minutes for vagrant)
- Reduced disk usage thanks to COW
- Reduced memory footprint thanks to KSM

Warnings about **libvirt-coreos** use case

The primary goal of the `libvirt-coreos` cluster provider is to deploy a multi-node Kubernetes cluster on local VMs as fast as possible and to be as light as possible in term of resources used.

In order to achieve that goal, its deployment is very different from the "standard production deployment" method used on other providers. This was done on purpose in order to implement some optimizations made possible by the fact that we know that all VMs will be running on the same physical machine.

The `libvirt-coreos` cluster provider doesn't aim at being production look-alike.

Another difference is that no security is enforced on `libvirt-coreos` at all. For example,

- Kube API server is reachable via a clear-text connection (no SSL);
- Kube API server requires no credentials;
- etcd access is not protected;
- Kubernetes secrets are not protected as securely as they are on production environments;
- etc.

So, a k8s application developer should not validate its interaction with Kubernetes on `libvirt-coreos` because he might technically succeed in doing things that are prohibited on a production environment like:

- un-authenticated access to Kube API server;
- Access to Kubernetes private data structures inside etcd;
- etc.

On the other hand, `libvirt-coreos` might be useful for people investigating low level implementation of Kubernetes because debugging techniques like sniffing the network traffic or introspecting the etcd content are easier on `libvirt-coreos` than on a production deployment.

Prerequisites

1. Install dnsmasq
2. Install ebtables
3. Install qemu
4. Install libvirt
5. Install openssl

6. Enable and start the libvirt daemon, e.g.:
    1. `systemctl enable libvirtd && systemctl start libvirtd` # for systemd-based systems
    2. `/etc/init.d/libvirt-bin start` # for init.d-based systems
7. Grant libvirt access to your user¹
8. Check that your $HOME is accessible to the qemu user²

¹ Depending on your distribution, libvirt access may be denied by default or may require a password at each access.

You can test it with the following command:

```
virsh -c qemu:///system pool-list
```

If you have access error messages, please read https://libvirt.org/acl.html and https://libvirt.org/aclpolkit.html.

In short, if your libvirt has been compiled with Polkit support (ex: Arch, Fedora 21), you can create `/etc/polkit-1/rules.d/50-org.libvirt.unix.manage.rules` as follows to grant full access to libvirt to `$USER`

```
sudo /bin/sh -c "cat - > /etc/polkit-1/rules.d/50-org.libvirt.unix.manage.rules" <
```

```
polkit.addRule(function(action, subject) {
        if (action.id == "org.libvirt.unix.manage" &&
            subject.user == "$USER") {
                return polkit.Result.YES;
                polkit.log("action=" + action);
                polkit.log("subject=" + subject);
        }
});
EOF
```

If your libvirt has not been compiled with Polkit (ex: Ubuntu 14.04.1 LTS), check the permissions on the libvirt unix socket:

```
$ ls -l /var/run/libvirt/libvirt-sock
srwxrwx--- 1 root libvirtd 0 févr. 12 16:03 /var/run/libvirt/libvirt-sock

$ usermod -a -G libvirtd $USER
# $USER needs to logout/login to have the new group be taken into account
```

(Replace `$USER` with your login name)

² Qemu will run with a specific user. It must have access to the VMs drives

All the disk drive resources needed by the VM (CoreOS disk image, Kubernetes binaries, cloud-init files, etc.) are put inside `./cluster/libvirt-coreos/libvirt_storage_pool`.

As we're using the `qemu:///system` instance of libvirt, qemu will run with a specific `user:group` distinct from your user. It is configured in `/etc/libvirt/qemu.conf`. That qemu user must have access to that libvirt storage pool.

If your `$HOME` is world readable, everything is fine. If your $HOME is private, `cluster/kube-up.sh` will fail with an error message like:

```
error: Cannot access storage file '$HOME/.../kubernetes/cluster/libvirt-coreos/lib
```

In order to fix that issue, you have several possibilities:

- set `POOL_PATH` inside `cluster/libvirt-coreos/config-default.sh` to a directory:
  - backed by a filesystem with a lot of free disk space
  - writable by your user;
  - accessible by the qemu user.
- Grant the qemu user access to the storage pool.

On Arch:

```
setfacl -m g:kvm:--x ~
```

Setup

By default, the libvirt-coreos setup will create a single Kubernetes master and 3 Kubernetes nodes. Because the VM drives use Copy-on-Write and because of memory ballooning and KSM, there is a lot of resource over-allocation.

There is both an automated way and a manual, customizable way of setting up libvirt Kubernetes clusters on CoreOS.

Automated setup

There is an automated setup script on https://get.k8s.io that will download the tarball for Kubernetes and spawn a Kubernetes cluster on a local CoreOS instances that the script creates. To run this script, use wget or curl with the KUBERNETES_PROVIDER environment variable set to libvirt-coreos:

```
export KUBERNETES_PROVIDER=libvirt-coreos; wget -q -O - https://get.k8s.io | bash
```

Here is the curl version of this command:

```
export KUBERNETES_PROVIDER=libvirt-coreos; curl -sS https://get.k8s.io | bash
```

This script downloads and unpacks the tarball, then spawns a Kubernetes cluster on CoreOS instances with the following characteristics:

- Total of 4 KVM/QEMU instances
- 1 instance acting as a Kubernetes master node
- 3 instances acting as minion nodes

If you'd like to run this cluster with customized settings, follow the manual setup instructions.

Manual setup

To start your local cluster, open a shell and run:

```
cd kubernetes

export KUBERNETES_PROVIDER=libvirt-coreos
cluster/kube-up.sh
```

The `KUBERNETES_PROVIDER` environment variable tells all of the various cluster management scripts which variant to use. If you forget to set this, the assumption is you are running on Google Compute Engine.

The `NUM_NODES` environment variable may be set to specify the number of nodes to start. If it is not set, the number of nodes defaults to 3.

The `KUBE_PUSH` environment variable may be set to specify which Kubernetes binaries must be deployed on the cluster. Its possible values are:

- `release` (default if `KUBE_PUSH` is not set) will deploy the binaries of `_output/release-tars/kubernetes-server-….tar.gz` . This is built with `make release` or `make release-skip-tests` .
- `local` will deploy the binaries of `_output/local/go/bin` . These are built with `make` .

Management

You can check that your machines are there and running with:

```
$ virsh -c qemu:///system list
 Id    Name                           State
----------------------------------------------------
 15    kubernetes_master              running
 16    kubernetes_node-01             running
 17    kubernetes_node-02             running
 18    kubernetes_node-03             running
 ```

You can check that the Kubernetes cluster is working with:

```shell
$ kubectl get nodes
NAME                STATUS     AGE      VERSION
192.168.10.2        Ready      4h       v1.6.0+fff5156
192.168.10.3        Ready      4h       v1.6.0+fff5156
192.168.10.4        Ready      4h       v1.6.0+fff5156
```

The VMs are running CoreOS. Your ssh keys have already been pushed to the VM. (It looks for ~/.ssh/id_*.pub) The user to use to connect to the VM is `core`. The IP to connect to the master is 192.168.10.1. The IPs to connect to the nodes are 192.168.10.2 and onwards.

Connect to `kubernetes_master`:

```
ssh core@192.168.10.1
```

Connect to `kubernetes_node-01`:

```
ssh core@192.168.10.2
```

Interacting with your Kubernetes cluster with the **kube-*** scripts.

All of the following commands assume you have set `KUBERNETES_PROVIDER` appropriately:

```
export KUBERNETES_PROVIDER=libvirt-coreos
```

Bring up a libvirt-CoreOS cluster of 5 nodes

```
NUM_NODES=5 cluster/kube-up.sh
```

Destroy the libvirt-CoreOS cluster

```
cluster/kube-down.sh
```

Update the libvirt-CoreOS cluster with a new Kubernetes release produced by `make release` or `make release-skip-tests`:

```
cluster/kube-push.sh
```

Update the libvirt-CoreOS cluster with the locally built Kubernetes binaries produced by `make`:

```
KUBE_PUSH=local cluster/kube-push.sh
```

Interact with the cluster

```
kubectl ...
```

Troubleshooting

!!! Cannot find kubernetes-server-linux-amd64.tar.gz

Build the release tarballs:

```
make release
```

Can’t find virsh in PATH, please fix and retry.

Install libvirt

On Arch:

```
pacman -S qemu libvirt
```

On Ubuntu 14.04:

```
aptitude install qemu-system-x86 libvirt-bin
```

On Fedora 21:

```
yum install qemu libvirt
```

error: Failed to connect socket to ‘/var/run/libvirt/libvirt-sock’: No such file or directory

Start the libvirt daemon

On Arch:

```
systemctl start libvirtd virtlogd.socket
```

The `virtlogd.socket` is not started with the libvirtd daemon. If you enable the `libvirtd.service` it is linked and started automatically on the next boot.

On Ubuntu 14.04:

```
service libvirt-bin start
```

error: Failed to connect socket to '/var/run/libvirt/libvirt-sock': Permission denied

Fix libvirt access permission (Remember to adapt `$USER`)

On Arch and Fedora 21:

```
cat > /etc/polkit-1/rules.d/50-org.libvirt.unix.manage.rules <<EOF
```

```
polkit.addRule(function(action, subject) {
        if (action.id == "org.libvirt.unix.manage" &&
            subject.user == "$USER") {
                return polkit.Result.YES;
                polkit.log("action=" + action);
                polkit.log("subject=" + subject);
        }
});
EOF
```

On Ubuntu:

```
usermod -a -G libvirtd $USER
```

error: Out of memory initializing network (virsh net-create…)

Ensure libvirtd has been restarted since ebtables was installed.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
libvirt/KVM	CoreOS	CoreOS	libvirt/KVM	docs		Community (@lhuard1A)

For support level information on all solutions, see the Table of solutions chart.

oVirt

What is oVirt

oVirt is a virtual datacenter manager that delivers powerful management of multiple virtual machines on multiple hosts. Using KVM and libvirt, oVirt can be installed on Fedora, CentOS, or Red Hat Enterprise Linux hosts to set up and manage your virtual data center.

oVirt Cloud Provider Deployment

The oVirt cloud provider allows to easily discover and automatically add new VM instances as nodes to your Kubernetes cluster. At the moment there are no community-supported or pre-loaded VM images including Kubernetes but it is possible to import or install Project Atomic (or Fedora) in a VM to generate a template. Any other distribution that includes Kubernetes may work as well.

It is mandatory to install the ovirt-guest-agent in the guests for the VM ip address and hostname to be reported to ovirt-engine and ultimately to Kubernetes.

Once the Kubernetes template is available it is possible to start instantiating VMs that can be discovered by the cloud provider.

Using the oVirt Cloud Provider

The oVirt Cloud Provider requires access to the oVirt REST-API to gather the proper information, the required credential should be specified in the `ovirt-cloud.conf` file:

```
[connection]
uri = https://localhost:8443/ovirt-engine/api
username = admin@internal
password = admin
```

In the same file it is possible to specify (using the `filters` section) what search query to use to identify the VMs to be reported to Kubernetes:

```
[filters]
# Search query used to find nodes
vms = tag=kubernetes
```

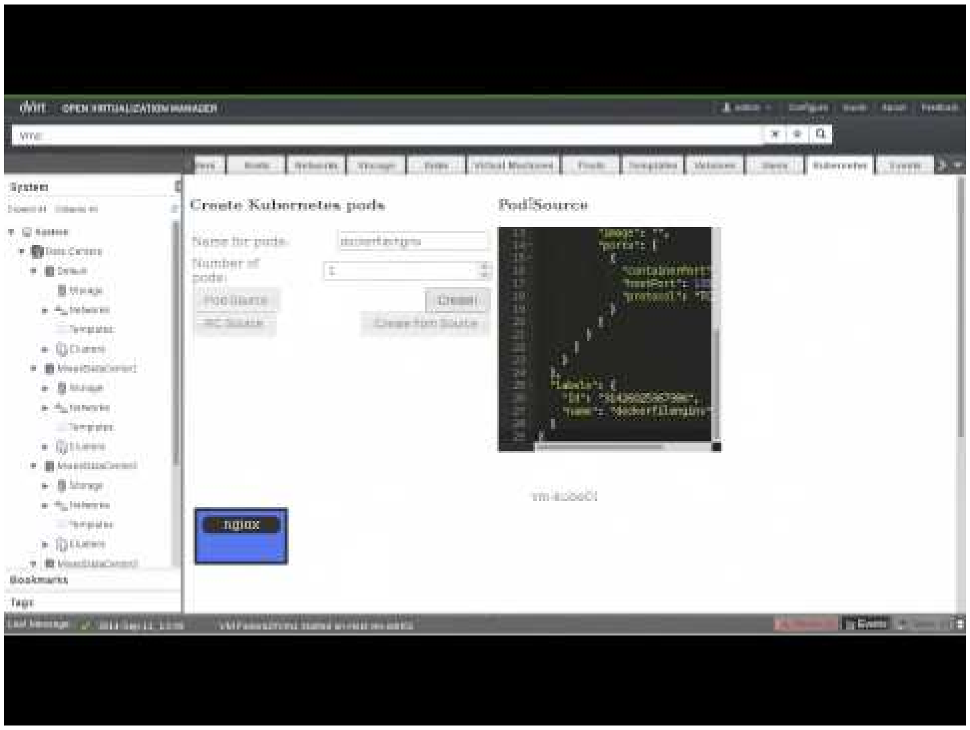
In the above example all the VMs tagged with the `kubernetes` label will be reported as nodes to Kubernetes.

The `ovirt-cloud.conf` file then must be specified in kube-controller-manager:

```
kube-controller-manager ... --cloud-provider=ovirt --cloud-config=/path/to/ovirt-c
```

oVirt Cloud Provider Screencast

This short screencast demonstrates how the oVirt Cloud Provider can be used to dynamically add VMs to your Kubernetes cluster.



Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
oVirt				[docs]		Community ([@simon3z])

For support level information on all solutions, see the [Table of solutions] chart.

OpenStack Heat

Getting started with OpenStack

Note: The `openstack-heat` provider that this guide uses is deprecated as of Kubernetes v1.8 and will be removed in a future release.

This guide will take you through the steps of deploying Kubernetes to Openstack using `kube-up.sh`. The primary mechanisms for this are OpenStack Heat and the SaltStack distributed with Kubernetes.

The default OS is CentOS 7, this has not been tested on other operating systems.

This guide assumes you have access to a working OpenStack cluster with the following features:

- Nova

- Neutron
- Swift
- Glance
- Heat
- DNS resolution of instance names

By default this provider provisions 4 `m1.medium` instances. If you do not have resources available, please see the Set additional configuration values section for information on reducing the footprint of your cluster.

Pre-Requisites

If you already have the required versions of the OpenStack CLI tools installed and configured, you can move on to the Starting a cluster section.

Install OpenStack CLI tools

```
sudo pip install -U --force 'python-openstackclient==3.11.0'
sudo pip install -U --force 'python-heatclient==1.10.0'
sudo pip install -U --force 'python-swiftclient==3.3.0'
sudo pip install -U --force 'python-glanceclient==2.7.0'
sudo pip install -U --force 'python-novaclient==9.0.1'
```

Configure Openstack CLI tools

Please talk to your local OpenStack administrator for an `openrc.sh` file.

Once you have that file, source it into your environment by typing

```
. ~/path/to/openrc.sh
```

This provider will consume the correct variables to talk to OpenStack and turn-up the Kubernetes cluster.

Otherwise, you must set the following appropriately:

```
export OS_USERNAME=username
export OS_PASSWORD=password
export OS_TENANT_NAME=projectName
export OS_AUTH_URL=https://identityHost:portNumber/v2.0
export OS_TENANT_ID=tenantIDString
export OS_REGION_NAME=regionName
```

Set additional configuration values

In addition, here are some commonly changed variables specific to this provider, with example values. Under most circumstances you will not have to change these. Please see the files in the next section for a full list of options.

```
export STACK_NAME=KubernetesStack
export NUMBER_OF_MINIONS=3
export MAX_NUMBER_OF_MINIONS=3
export MASTER_FLAVOR=m1.small
export MINION_FLAVOR=m1.small
export EXTERNAL_NETWORK=public
export DNS_SERVER=8.8.8.8
export IMAGE_URL_PATH=http://cloud.centos.org/centos/7/images
export IMAGE_FILE=CentOS-7-x86_64-GenericCloud-1510.qcow2
export SWIFT_SERVER_URL=http://192.168.123.100:8080
export ENABLE_PROXY=false
```

Manually overriding configuration values

If you do not have your environment variables set, or do not want them consumed, modify the variables in the following files under `cluster/openstack-heat`:

- **config-default.sh** Sets all parameters needed for heat template.
- **config-image.sh** Sets parameters needed to download and create new OpenStack image via glance.
- **openrc-default.sh** Sets environment variables for communicating to OpenStack. These are consumed by the cli tools (heat, glance, swift, nova).
- **openrc-swift.sh** Some OpenStack setups require the use of separate swift credentials. Put those credentials in this file.

Please see the contents of these files for documentation regarding each variable's function.

Starting a cluster

Once you've installed the OpenStack CLI tools and have set your OpenStack environment variables, issue this command:

```
export KUBERNETES_PROVIDER=openstack-heat; curl -sS https://get.k8s.io | bash
```

Alternatively, you can download a Kubernetes release of version 1.3 or higher and extract the archive. To start your cluster, open a shell and run:

```
cd kubernetes # Or whichever path you have extracted the release to
KUBERNETES_PROVIDER=openstack-heat ./cluster/kube-up.sh
```

Or, if you are working from a checkout of the Kubernetes code base, and want to build/test from source:

```
cd kubernetes # Or whatever your checkout root directory is called
make clean
make quick-release
KUBERNETES_PROVIDER=openstack-heat ./cluster/kube-up.sh
```

Inspect your cluster

Once kube-up is finished, your cluster should be running:

```
./cluster/kubectl.sh get cs
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-1               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
```

You can also list the nodes in your cluster:

```
./cluster/kubectl.sh get nodes
NAME                            STATUS    AGE     VERSION
kubernetesstack-node-ojszyjtr   Ready     42m     v1.6.0+fff5156
kubernetesstack-node-tzotzcbp   Ready     46m     v1.6.0+fff5156
kubernetesstack-node-uah8pkju   Ready     47m     v1.6.0+fff5156
```

Being a new cluster, there will be no pods or replication controllers in the default namespace:

```
./cluster/kubectl.sh get pods
./cluster/kubectl.sh get replicationcontrollers
```

You are now ready to create Kubernetes objects.

Using your cluster

For a simple test, issue the following command:

```
./cluster/kubectl.sh run nginx --image=nginx --generator=run-pod/v1
```

Soon, you should have a running nginx pod:

```
./cluster/kubectl.sh get pods
NAME      READY     STATUS     RESTARTS    AGE
nginx     1/1       Running    0           5m
```

Once the nginx pod is running, use the port-forward command to set up a proxy from your machine to the pod.

```
./cluster/kubectl.sh port-forward nginx 8888:80
```

You should now see nginx on http://localhost:8888.

For more complex examples please see the examples directory.

Administering your cluster with Openstack

You can manage the nodes in your cluster using the OpenStack CLI Tools.

First, set your environment variables:

```
. cluster/openstack-heat/config-default.sh
. cluster/openstack-heat/openrc-default.sh
```

To get all information about your cluster, use heat:

```
openstack stack show $STACK_NAME
```

To see a list of nodes, use nova:

```
nova list --name=$STACK_NAME
```

See the OpenStack CLI Reference for more details.

Salt

The OpenStack-Heat provider uses a standalone Salt configuration. It only uses Salt for bootstrapping the machines and creates no salt-master and does not auto-start the salt-minion service on the nodes.

SSHing to your nodes

Your public key was added during the cluster turn-up, so you can easily ssh to them for troubleshooting purposes.

```
ssh minion@IP_ADDRESS
```

Cluster deployment customization examples

You may find the need to modify environment variables to change the behaviour of kube-up. Here are some common scenarios:

Proxy configuration

If you are behind a proxy, and have your local environment variables setup, you can use these variables to setup your Kubernetes cluster:

```
ENABLE_PROXY=true KUBERNETES_PROVIDER=openstack-heat ./cluster/kube-up.sh
```

Setting different Swift URL

Some deployments differ from the default Swift URL:

```
 SWIFT_SERVER_URL="http://10.100.0.100:8080" KUBERNETES_PROVIDER=openstack-heat ./
```

Public network name.

Sometimes the name of the public network differs from the default `public`:

```
EXTERNAL_NETWORK="network_external" KUBERNETES_PROVIDER=openstack-heat ./cluster/k
```

Spinning up additional clusters.

You may want to spin up another cluster within your OpenStack project. Use the `$STACK_NAME` variable to accomplish this.

```
STACK_NAME=k8s-cluster-2 KUBERNETES_PROVIDER=openstack-heat ./cluster/kube-up.sh
```

For more configuration examples, please browse the files mentioned in the Configuration section.

Tearing down your cluster

To bring down your cluster, issue the following command:

```
KUBERNETES_PROVIDER=openstack-heat ./cluster/kube-down.sh
```

If you have changed the default `$STACK_NAME`, you must specify the name. Note that this will not remove any Cinder volumes created by Kubernetes.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
OpenStack Heat	Saltstack	CentOS	Neutron + flannel hostgw	docs		Community (@FujitsuEnablingSoftwareTechnologyGmbH)

For support level information on all solutions, see the Table of solutions chart.

Running Kubernetes with rkt

This document describes how to run Kubernetes using rkt as the container runtime.

Note: This document describes how to use what is known as “rktnetes”. In future, Kubernetes will support the rkt runtime through the Container Runtime Interface (CRI). At present the rkt shim for the CRI is considered “experimental”, but if you wish to use it you will find instructions in the kubeadm reference.

Prerequisites

- Systemd must be installed and enabled. The minimum systemd version required for Kubernetes v1.3 is `219`. Systemd is used to monitor and manage the pods on each node.

- Install the latest rkt release. The minimum rkt version required is v1.13.0. The CoreOS Linux alpha channel ships with a recent rkt release, and you can easily upgrade rkt on CoreOS, if necessary.
- The rkt API service must be running on the node.
- You will need kubelet installed on the node, and it's recommended that you run kube-proxy on all nodes. This document describes how to set the parameters for kubelet so that it uses rkt as the runtime.

Pod networking in rktnetes

Kubernetes CNI networking

You can configure Kubernetes pod networking with the usual Container Network Interface (CNI) network plugins by setting the kubelet's `--network-plugin` and `--network-plugin-dir` options appropriately. Configured in this fashion, the rkt container engine will be unaware of network details, and expects to connect pods to the provided subnet.

kubenet: Google Compute Engine (GCE) network

The `kubenet` plugin can be selected with the kubelet option `--network-plugin=kubenet`. This plugin is currently only supported on GCE. When using kubenet, Kubernetes CNI creates and manages the network, and rkt is provided with a subnet from a bridge device connected to the GCE network.

rkt contained network

Rather than delegating pod networking to Kubernetes, rkt can configure connectivity directly with its own *contained network* on a subnet provided by a bridge device, the flannel SDN, or another CNI plugin. Configured this way, rkt looks in its config directories, usually `/etc/rkt/net.d`, to discover the CNI configuration and invoke the appropriate plugins to create the pod network.

rkt contained network with bridge

The *contained network* is rkt's default, so you can leave the kubelet's `--network-plugin` option empty to select this network. The contained network can be backed by any CNI plugin. With the *contained network*, rkt will attempt to join pods to a network named `rkt.kubernetes.io`, so this network name must be used for whatever desired CNI configuration.

When using the contained network, create a network configuration file beneath the rkt network config directory that defines how to create this `rkt.kubernetes.io` network in your environment. This example sets up a bridge device with the `bridge` CNI plugin:

```
$ cat <<EOF >/etc/rkt/net.d/k8s_network_example.conf
{
  "name": "rkt.kubernetes.io",
  "type": "bridge",
  "bridge": "mybridge",
  "mtu": 1460,
  "addIf": "true",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.22.0.0/16",
    "gateway": "10.22.0.1",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
EOF
```

rkt contained network with flannel

While it is recommended to operate flannel through the Kubernetes CNI support, you can alternatively configure the flannel plugin directly to provide the subnet for rkt's contained network. An example CNI/flannel config file looks like this:

```
$ cat <<EOF >/etc/rkt/net.d/k8s_flannel_example.conf
{
    "name": "rkt.kubernetes.io",
    "type": "flannel",
    "delegate": {
        "isDefaultGateway": true
    }
}
EOF
```

For more information on flannel configuration, see the CNI/flannel README.

Contained network caveats:

- You must create an appropriate CNI configuration file with a network name of `rkt.kubernetes.io`.
- The downwards API and environment variable substitution will not contain the pod IP address.
- The `/etc/hosts` file will not contain the pod's own hostname, although `/etc/hostname` is populated.

Running rktnetes

Spin up a local Kubernetes cluster with the rkt runtime

To use rkt as the container runtime in a local Kubernetes cluster, supply the following flags to the kubelet:

- `--container-runtime=rkt` Set the node's container runtime to rkt.
- `--rkt-api-endpoint=HOST:PORT` Set the endpoint of the rkt API service. Default: `localhost:15441`.
- `--rkt-path=PATH_TO_RKT_BINARY` Set the path of the rkt binary. Optional. If empty, look for `rkt` in `$PATH`.
- `--rkt-stage1-image=STAGE1` Set the name of the stage1 image, e.g. `coreos.com/rkt/stage1-coreos`. Optional. If not set, the default Linux kernel software isolation stage1 is used.

If you are using the hack/local-up-cluster.sh script to launch the cluster, you can edit the environment variables `CONTAINER_RUNTIME`, `RKT_PATH`, and `RKT_STAGE1_IMAGE` to set these flags. `RKT_PATH` and `RKT_STAGE1_IMAGE` are optional if `rkt` is in your $PATH` with appropriate configuration.

```
$ export CONTAINER_RUNTIME=rkt
$ export RKT_PATH=<rkt_binary_path>
$ export RKT_STAGE1_IMAGE=<stage1-name>
```

Now you can launch the cluster using the `local-up-cluster.sh` script:

```
$ hack/local-up-cluster.sh
```

We are also working on getting rkt working as the container runtime in minikube.

Launch a rktnetes cluster on Google Compute Engine (GCE)

This section outlines using the `kube-up` script to launch a CoreOS/rkt cluster on GCE.

Specify the OS distribution, the GCE distributor's master project, and the instance images for the Kubernetes master and nodes. Set the `KUBE_CONTAINER_RUNTIME` to `rkt`:

```
$ export KUBE_OS_DISTRIBUTION=coreos
$ export KUBE_GCE_MASTER_PROJECT=coreos-cloud
$ export KUBE_GCE_MASTER_IMAGE=<image_id>
$ export KUBE_GCE_NODE_PROJECT=coreos-cloud
$ export KUBE_GCE_NODE_IMAGE=<image_id>
$ export KUBE_CONTAINER_RUNTIME=rkt
```

Optionally, set the version of rkt by setting `KUBE_RKT_VERSION`:

```
$ export KUBE_RKT_VERSION=1.13.0
```

Optionally, select an alternative stage1 isolator for the container runtime by setting `KUBE_RKT_STAGE1_IMAGE`:

```
$ export KUBE_RKT_STAGE1_IMAGE=<stage1-name>
```

Then you can launch the cluster with:

```
$ cluster/kube-up.sh
```

Launch a rktnetes cluster on AWS

The `kube-up` script is not yet supported on AWS. Instead, we recommend following the Kubernetes on AWS guide to launch a CoreOS Kubernetes cluster on AWS, then setting kubelet options as above.

Deploy apps to the cluster

After creating the cluster, you can start deploying applications. For an introductory example, deploy a simple nginx web server. Note that this example did not have to be modified for use with a “rktnetes” cluster. More examples can be found in the Kubernetes examples directory.

Modular isolation with interchangeable stage1 images

rkt executes containers in an interchangeable isolation environment. This facility is called the *stage1 image*. There are currently three supported rkt stage1 images:

- `systemd-nspawn` stage1, the default. Isolates running containers with Linux kernel namespaces and cgroups in a manner similar to the default container runtime.
- `KVM` stage1, runs containers inside a KVM hypervisor-managed virtual machine. Experimental in the Kubernetes v1.3 release.
- `fly stage1`, which isolates containers with only a `chroot`, giving host-level access to mount and network namespaces for specially-privileged utilities.

In addition to the three provided stage1 images, you can create your own for specific isolation requirements. If no configuration is set, the default stage1 is used. There are two ways to select a different stage1; either per-node, or per-pod:

- Set the kubelet’s `--rkt-stage1-image` flag, which tells the kubelet the stage1 image to use for every pod on the node. For example, `--rkt-stage1-image=coreos/rkt/stage1-coreos` selects the default systemd-nspawn stage1.
- Set the annotation `rkt.alpha.kubernetes.io/stage1-name-override` to override the stage1 used to execute a given pod. This allows for mixing different container isolation mechanisms on

the same cluster or on the same node. For example, the following (shortened) pod manifest will run its pod with the `fly stage1` to give the application – the `kubelet` in this case – access to the host's namespace:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubelet
  namespace: kube-system
  labels:
    k8s-app: kubelet
  annotations:
    rkt.alpha.kubernetes.io/stage1-name-override: coreos.com/rkt/stage1-fly
spec:
  containers:
  - name: kubelet
    image: quay.io/coreos/hyperkube:v1.3.0-beta.2_coreos.0
    command:
    - kubelet
    - --api-servers=127.0.0.1:8080
    - --config=/etc/kubernetes/manifests
    - --allow-privileged
    - --kubeconfig=/etc/kubernetes/kubeconfig
    securityContext:
      privileged: true
[...]
```

Notes on using different stage1 images

Setting the stage1 annotation could potentially give the pod root privileges. Because of this, the `privileged` boolean in the pod's `securityContext` must be set to `true`.

Use rkt's *contained network* with the KVM stage1, because the CNI plugin driver does not yet fully support the hypervisor-based runtime.

Known issues and differences between rkt and Docker

rkt and the default node container engine have very different designs, as do rkt's native ACI and the Docker container image format. Users may experience different behaviors when switching from one container engine to the other. More information can be found in the Kubernetes rkt notes.

Troubleshooting

Here are a few tips for troubleshooting Kubernetes with the rkt container engine:

Check rkt pod status

To check the status of running pods, use the rkt subcommands `rkt list`, `rkt status`, and `rkt image list`. See the rkt commands documentation for more information about rkt subcommands.

Check journal logs

Check a pod's log using `journalctl` on the node. Pods are managed and named as systemd units. The pod's unit name is formed by concatenating a `k8s_` prefix with the pod UUID, in a format like `k8s_${RKT_UUID}`. Find the pod's UUID with `rkt list` to assemble its service name, then ask journalctl for the logs:

```
$ sudo journalctl -u k8s_ad623346
```

Log verbosity

By default, the log verbosity level is 2. In order to see more log messages related to rkt, set this level to 4 or above. For a local cluster, set the environment variable: `LOG_LEVEL=4`.

Check Kubernetes events and logs.

Kubernetes provides various tools for troubleshooting and examination. More information can be found in the app troubleshooting guide.

Known Issues when Using rkt

The following features either are not supported or have large caveats when using the rkt container runtime. Increasing support for these items and others, including reasonable feature parity with the default container engine, is planned through future releases.

Non-existent host volume paths

When mounting a host volume path that does not exist, rkt will error out. Under the Docker runtime, an empty directory will be created at the referenced path.

An example of a pod which will error out:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: mount-dne
  name: mount-dne
spec:
  volumes:
  - name: does-not-exist
    hostPath:
      path: /does/not/exist
  containers:
    - name: exit
      image: busybox
      command: ["sh", "-c", "ls /test; sleep 60"]
      volumeMounts:
      - mountPath: /test
        name: does-not-exist
```

Also note that if `subPath` is specified in the container's volumeMounts and the `subPath` doesn't exist in the corresponding volume, the pod execution will fail as well.

Kubectl attach

The `kubectl attach` command does not work under the rkt container runtime. Because of this, some flags in `kubectl run` are not supported, including:

- `--attach=true`
- `--leave-stdin-open=true`
- `--rm=true`

Port forwarding for kvm and fly stage1s

`kubectl port-forward` is not supported for pods that are executed with `stage1-kvm` or `stage1-fly`.

Volume relabeling

Currently rkt supports only *per-pod* volume relabeling. After relabeling, the mounted volume is shared by all Containers in the pod. There is not yet a way to make the relabeled volume accessible to only one, or some subset, of Containers in the pod. Kubernetes issue # 28187 has the details.

kubectl get logs

Under rktnetes, `kubectl get logs` currently cannot get logs from applications that write them to directly to `/dev/stdout`. Currently such log messages are printed on the node's console.

Init Containers

Init Containers are currently not supported.

Container restart back-off

Exponential restart back-off for a failing container is currently not supported.

Experimental NVIDIA GPU support

The `--experimental-nvidia-gpus` flag, and related GPU features are not supported.

QoS Classes

Under rkt, QoS classes do not adjust the `OOM Score` of Containers as occurs under Docker.

HostPID and HostIPC namespaces

Setting the hostPID or hostIPC flags on a pod is not supported.

For example, the following pod will not run correctly:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: host-ipc-pid
  name: host-ipc-pid
spec:
  hostIPC: true
  hostPID: true
  containers:
    ...
```

On the other hand, when running the pod with stage1-fly, the pod will be run in the host namespace.

Container image updates (patch)

Patching a pod to change the image will result in the entire pod restarting, not just the container that was changed.

ImagePullPolicy ‘Always’

When the container’s image pull policy is `Always`, rkt will always pull the image from remote even if the image has not changed at all. This can add significant latency for large images. The issue is tracked by rkt upstream at #2937.

Kubernetes on Mesos

About Kubernetes on Mesos

Mesos allows dynamic sharing of cluster resources between Kubernetes and other first-class Mesos frameworks such as HDFS, Spark, and Chronos. Mesos also ensures applications from different frameworks running on your cluster are isolated and that resources are allocated fairly among them.

Mesos clusters can be deployed on nearly every IaaS cloud provider infrastructure or in your own physical datacenter. Kubernetes on Mesos runs on-top of that and therefore allows you to easily move Kubernetes workloads from one of these environments to the other.

This tutorial will walk you through setting up Kubernetes on a Mesos cluster. It provides a step by step walk through of adding Kubernetes to a Mesos cluster and starting your first pod with an nginx webserver.

NOTE: There are known issues with the current implementation and support for centralized logging and monitoring is not yet available. Please file an issue against the kubernetes-mesos project if you have problems completing the steps below.

Further information is available in the Kubernetes on Mesos contrib directory.

Prerequisites

- Understanding of Apache Mesos

- A running Mesos cluster on Google Compute Engine
- A VPN connection to the cluster
- A machine in the cluster which should become the Kubernetes *master node* with:
    - Go (see here for required versions)
    - make (i.e. build-essential)
    - Docker

Note: You *can*, but you *don't have to* deploy Kubernetes-Mesos on the same machine the Mesos master is running on.

Deploy Kubernetes-Mesos

Log into the future Kubernetes *master node* over SSH, replacing the placeholder below with the correct IP address.

```
ssh jclouds@${ip_address_of_master_node}
```

Build Kubernetes-Mesos.

```
git clone https://github.com/kubernetes-incubator/kube-mesos-framework
cd kube-mesos-framework
make
```

Set some environment variables. The internal IP address of the master may be obtained via `hostname -i`.

```
export KUBERNETES_MASTER_IP=$(hostname -i)
export KUBERNETES_MASTER=http://${KUBERNETES_MASTER_IP}:8888
```

Note that KUBERNETES_MASTER is used as the api endpoint. If you have existing `~/.kube/config` and point to another endpoint, you need to add option `--server=${KUBERNETES_MASTER}` to kubectl in later steps.

Deploy etcd

Start etcd and verify that it is running:

```
sudo docker run -d --hostname $(uname -n) --name etcd \
  -p 4001:4001 -p 7001:7001 quay.io/coreos/etcd:v2.2.1 \
  --listen-client-urls http://0.0.0.0:4001 \
  --advertise-client-urls http://${KUBERNETES_MASTER_IP}:4001
```

```
$ sudo docker ps
CONTAINER ID   IMAGE                        COMMAND    CREATED   STATUS   PORTS
fd7bac9e2301   quay.io/coreos/etcd:v2.2.1   "/etcd"    5s ago    Up 3s    2379/tcp,
```

It's also a good idea to ensure your etcd instance is reachable by testing it

```
curl -L http://${KUBERNETES_MASTER_IP}:4001/v2/keys/
```

If connectivity is OK, you will see an output of the available keys in etcd (if any).

Start Kubernetes-Mesos Services

Update your PATH to more easily run the Kubernetes-Mesos binaries:

```
export PATH="$(pwd)/_output/local/go/bin:$PATH"
```

Identify your Mesos master: depending on your Mesos installation this is either a `host:port` like `mesos-master:5050` or a ZooKeeper URL like `zk://zookeeper:2181/mesos`. In order to let Kubernetes survive Mesos master changes, the ZooKeeper URL is recommended for production environments.

```
export MESOS_MASTER=<host:port or zk:// url>
```

Create a cloud config file `mesos-cloud.conf` in the current directory with the following contents:

```
$ cat <<EOF >mesos-cloud.conf
[mesos-cloud]
        mesos-master        = ${MESOS_MASTER}
EOF
```

Now start the kubernetes-mesos API server, controller manager, and scheduler on the master node:

```
$ km apiserver \
  --address=${KUBERNETES_MASTER_IP} \
  --etcd-servers=http://${KUBERNETES_MASTER_IP}:4001 \
  --service-cluster-ip-range=10.10.10.0/24 \
  --port=8888 \
  --cloud-provider=mesos \
  --cloud-config=mesos-cloud.conf \
  --secure-port=0 \
  --v=1 >apiserver.log 2>&1 &

$ km controller-manager \
  --master=${KUBERNETES_MASTER_IP}:8888 \
  --cloud-provider=mesos \
  --cloud-config=./mesos-cloud.conf  \
  --v=1 >controller.log 2>&1 &

$ km scheduler \
  --address=${KUBERNETES_MASTER_IP} \
  --mesos-master=${MESOS_MASTER} \
  --etcd-servers=http://${KUBERNETES_MASTER_IP}:4001 \
  --mesos-user=root \
  --api-servers=${KUBERNETES_MASTER_IP}:8888 \
  --cluster-dns=10.10.10.10 \
  --cluster-domain=cluster.local \
  --v=2 >scheduler.log 2>&1 &
```

Disown your background jobs so that they'll stay running if you log out.

```
disown -a
```

Validate KM Services

Interact with the kubernetes-mesos framework via `kubectl`:

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

```
# NOTE: your service IPs will likely differ
$ kubectl get services
NAME             CLUSTER-IP       EXTERNAL-IP     PORT(S)        AGE
k8sm-scheduler   10.10.10.113     <none>          10251/TCP      1d
kubernetes       10.10.10.1       <none>          443/TCP        1d
```

Lastly, look for Kubernetes in the Mesos web GUI by pointing your browser to `http://<mesos-master-ip:port>`. Make sure you have an active VPN connection. Go to the Frameworks tab, and look for an active framework named "Kubernetes".

Spin up a pod

Write a JSON pod description to a local file:

```
$ cat <<EOPOD >nginx.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
EOPOD
```

Send the pod description to Kubernetes using the `kubectl` CLI:

```
$ kubectl create -f ./nginx.yaml
pod "nginx" created
```

Wait a minute or two while `dockerd` downloads the image layers from the internet. We can use the `kubectl` interface to monitor the status of our pod:

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          14s
```

Verify that the pod task is running in the Mesos web GUI. Click on the Kubernetes framework. The next screen should show the running Mesos task that started the Kubernetes pod.

Launching kube-dns

Kube-dns is an addon for Kubernetes which adds DNS-based service discovery to the cluster. For a detailed explanation see DNS in Kubernetes.

The kube-dns addon runs as a pod inside the cluster. The pod consists of three co-located containers:

- a local etcd instance
- the kube-dns DNS server

We assume that kube-dns will use

- the service IP `10.10.10.10`
- and the `cluster.local` domain.

Note that we have passed these two values already as parameter to the apiserver above.

A template for a replication controller spinning up the pod with the 3 containers can be found at cluster/addons/dns/kubedns-controller.yaml.in in the repository. The following steps are necessary in order to get a valid replication controller yaml file:

- replace `{{ pillar['dns_replicas'] }}` with `1`
- replace `{{ pillar['dns_domain'] }}` with `cluster.local.`
- add `--kube_master_url=${KUBERNETES_MASTER}` parameter to the kube2sky container command.

In addition the service template at cluster/addons/dns/kubedns-controller.yaml.in needs the following replacement:

- `{{ pillar['dns_server'] }}` with `10.10.10.10`.

To do this automatically:

```
sed -e "s/{{ pillar\['dns_replicas'\] }}/1/g;"\
"s,\(command = \"/kube2sky\"\),\\1\\"$'\n'"        - --kube_master_url=${KUBERNETE
"s/{{ pillar\['dns_domain'\] }}/cluster.local/g" \
  cluster/addons/dns/kubedns-controller.yaml.in > kubedns-controller.yaml
sed -e "s/{{ pillar\['dns_server'\] }}/10.10.10.10/g" \
  cluster/addons/dns/kubedns-svc.yaml.in > kubedns-svc.yaml
```

Now the kube-dns pod and service are ready to be launched:

```
kubectl create -f ./kubedns-controller.yaml
kubectl create -f ./kubedns-svc.yaml
```

Check with `kubectl get pods --namespace=kube-system` that 3/3 containers of the pods are eventually up and running. Note that the kube-dns pods run in the `kube-system` namespace, not in `default`.

To check that the new DNS service in the cluster works, we start a busybox pod and use that to do a DNS lookup. First create the `busybox.yaml` pod spec:

```
cat <<EOF >busybox.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
EOF
```

Then start the pod:

```
kubectl create -f ./busybox.yaml
```

When the pod is up and running, start a lookup for the Kubernetes master service, made available on 10.10.10.1 by default:

```
kubectl  exec busybox -- nslookup kubernetes
```

If everything works fine, you will get this output:

```
Server:    10.10.10.10
Address 1: 10.10.10.10

Name:      kubernetes
Address 1: 10.10.10.1
```

## Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Mesos/GCE				docs		Community (Kubernetes-Mesos Authors)

For support level information on all solutions, see the Table of solutions chart.

What next?

Try out some of the standard Kubernetes examples.

Read about Kubernetes on Mesos' architecture in the contrib directory.

NOTE: Some examples require Kubernetes DNS to be installed on the cluster. Future work will add instructions to this guide to enable support for Kubernetes DNS.

NOTE: Please be aware that there are known issues with the current Kubernetes-Mesos implementation.

Kubernetes on Mesos on Docker

The mesos/docker provider uses docker-compose to launch Kubernetes as a Mesos framework, running in docker with its dependencies (etcd & mesos).

Cluster Goals

- kubernetes development
- pod/service development
- demoing
- fast deployment
- minimal hardware requirements
- minimal configuration
- entry point for exploration
- simplified networking

- fast end-to-end tests
- local deployment

Non-Goals:

- high availability
- fault tolerance
- remote deployment
- production usage
- monitoring
- long running
- state persistence across restarts

Cluster Topology

The cluster consists of several docker containers linked together by docker-managed hostnames:

Component	Hostname	Description
docker-grand-ambassador		Proxy to allow circular hostname linking in docker
etcd	etcd	Key/Value store used by Mesos
Mesos Master	mesosmaster1	REST endpoint for interacting with Mesos
Mesos Slave (x2)	mesosslave1, mesosslave2	Mesos agents that offer resources and run framework executors (e.g. Kubernetes Kublets)
Kubernetes API Server	apiserver	REST endpoint for interacting with Kubernetes
Kubernetes Controller Manager	controller	
Kubernetes Scheduler	scheduler	Schedules container deployment by accepting Mesos offers

Prerequisites

Required:

- Git - version control system
- Docker CLI - container management command line client
- Docker Engine - container management daemon
  - On Mac, use Docker Machine
- Docker Compose - multi-container application orchestration

Optional:

- Virtual Box
  - Free x86 virtualization engine with a Docker Machine driver
- Golang - Go programming language
  - Required to build Kubernetes locally
- Make - Utility for building executables from source
  - Required to build Kubernetes locally with make

Install on Mac (Homebrew)

It's possible to install all of the above via Homebrew on a Mac.

Some steps print instructions for configuring or launching. Make sure each is properly set up before continuing to the next step.

```
brew install git
brew install caskroom/cask/brew-cask
brew cask install virtualbox
brew install docker
brew install docker-machine
brew install docker-compose
```

Install on Linux

Most of the above are available via apt and yum, but depending on your distribution, you may have to install via other means to get the latest versions.

It is recommended to use Ubuntu, simply because it best supports AUFS, used by docker to mount volumes. Alternate file systems may not fully support docker-in-docker.

In order to build Kubernetes, the current user must be in a docker group with sudo privileges. See the docker docs for instructions.

Docker Machine Config (Mac)

If on a Mac using docker-machine, the following steps will make the docker IPs (in the virtualbox VM) reachable from the host machine (Mac).

1. Create VM

   oracle-virtualbox

   ```
   docker-machine create --driver virtualbox kube-dev
   eval "$(docker-machine env kube-dev)"
   ```

2. Set the VM's host-only network to "promiscuous mode":

   oracle-virtualbox

   ```
   conf docker-machine stop kube-dev VBoxManage modifyvm kube-dev --nicpromisc2
   allow-all docker-machine start kube-dev
   ```

   This allows the VM to accept packets that were sent to a different IP.

   Since the host-only network routes traffic between VMs and the host, other VMs will also be able to access the docker IPs, if they have the following route.

3. Route traffic to docker through the docker-machine IP:

```
sudo route -n add -net 172.17.0.0 $(docker-machine ip kube-dev)
```

```
Since the docker-machine IP can change when the VM is restarted, this route may ne
To delete the route later: `sudo route delete 172.17.0.0`
```

Walkthrough

1. Checkout source

   ```
   shell git clone https://github.com/kubernetes/kubernetes cd kubernetes
   ```

   By default, that will get you the bleeding edge of master branch. You may want a release branch instead, if you have trouble with master.

2. Build binaries

   You'll need to build kubectl (CLI) for your local architecture and operating system and the rest of the server binaries for linux/amd64.

   Building a new release covers both cases:

   ```
   shell KUBERNETES_CONTRIB=mesos build/release.sh
   ```

   For developers, it may be faster to build locally.

3. [Optional] Build docker images

   The following docker images are built as part of `./cluster/kube-up.sh`, but it may make sense to build them manually the first time because it may take a while.

   1. Test image includes all the dependencies required for running e2e tests.

      ```
      shell ./cluster/mesos/docker/test/build.sh
      ```

      In the future, this image may be available to download. It doesn't contain anything specific to the current release, except its build dependencies.

   2. Kubernetes-Mesos image includes the compiled linux binaries.

      ```
      shell ./cluster/mesos/docker/km/build.sh
      ```

      This image needs to be built every time you recompile the server binaries.

4. [Optional] Configure Mesos resources

   By default, the mesos-slaves are configured to offer a fixed amount of resources (cpus, memory, disk, ports). If you want to customize these values, update the `MESOS_RESOURCES` environment variables in `./cluster/mesos/docker/docker-compose.yml`. If you delete the

`MESOS_RESOURCES` environment variables, the resource amounts will be auto-detected based on the host resources, which will over-provision by > 2x.

If the configured resources are not available on the host, you may want to increase the resources available to Docker Engine. You may have to increase you VM disk, memory, or cpu allocation. See the Docker Machine docs for details (Virtualbox)

5. Configure provider

   ```
   shell export KUBERNETES_PROVIDER=mesos/docker
   ```

   This tells cluster scripts to use the code within `cluster/mesos/docker`.

6. Create cluster

   ```
   shell ./cluster/kube-up.sh
   ```

   If you manually built all the above docker images, you can skip that step during kube-up:

   ```
   shell MESOS_DOCKER_SKIP_BUILD=true ./cluster/kube-up.sh
   ```

   After deploying the cluster, `~/.kube/config` will be created or updated to configure kubectl to target the new cluster.

7. Explore tutorials

   To learn more about Pods, Volumes, Labels, Services, and Replication Controllers, start with the Kubernetes Tutorials.

   To skip to a more advanced example, see the Guestbook Example

8. Destroy cluster

   ```
   shell ./cluster/kube-down.sh
   ```

Addons

The `kube-up` for the mesos/docker provider will automatically deploy KubeDNS and KubeUI addons as pods/services.

Check their status with:

```
./cluster/kubectl.sh get pods --namespace=kube-system
```

KubeUI

The web-based Kubernetes UI is accessible in a browser through the API Server proxy: `https://<apiserver>:6443/ui/`.

By default, basic-auth is configured with user `admin` and password `admin`.

The IP of the API Server can be found using `./cluster/kubectl.sh cluster-info`.

End To End Testing

Warning: e2e tests can take a long time to run. You may not want to run them immediately if you're just getting started.

While your cluster is up, you can run the end-to-end tests:

```
./cluster/test-e2e.sh
```

Notable parameters: - Increase the logging verbosity: `-v=2` - Run only a subset of the tests (regex matching): `-ginkgo.focus=<pattern>`

To build, deploy, test, and destroy, all in one command (plus unit & integration tests):

```
make test_e2e
```

Kubernetes CLI

When compiling from source, it's simpler to use the `./cluster/kubectl.sh` script, which detects your platform & architecture and proxies commands to the appropriate `kubectl` binary.

ex: `./cluster/kubectl.sh get pods`

Helpful scripts

- Kill all docker containers

  `shell docker ps -q -a | xargs docker rm -f`

- Clean up unused docker volumes

  `shell docker run -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker:/var/lib/docker --rm martin/docker-cleanup-volumes`

Build Locally

The steps above tell you how to build in a container, for minimal local dependencies. But if you have Go and Make installed you can build locally much faster:

```
KUBERNETES_CONTRIB=mesos make
```

However, if you're not on linux, you'll still need to compile the linux/amd64 server binaries:

```
KUBERNETES_CONTRIB=mesos build/run.sh hack/build-go.sh
```

The above two steps should be significantly faster than cross-compiling a whole new release for every supported platform (which is what `./build/release.sh` does).

Breakdown:

- `KUBERNETES_CONTRIB=mesos` - enables building of the contrib/mesos binaries
- `hack/build-go.sh` - builds the Go binaries for the current architecture (linux/amd64 when in a docker container)
- `make` - delegates to `hack/build-go.sh`
- `build/run.sh` - executes a command in the build container

- `build/release.sh` - cross compiles Kubernetes for all supported architectures and operating systems (slow)

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Mesos/Docker	custom	Ubuntu	Docker	docs		Community (Kubernetes-Mesos Authors)

For support level information on all solutions, see the Table of solutions chart.

Offline

Deploy a CoreOS running Kubernetes environment. This particular guide is made to help those in an OFFLINE system, whether for testing a POC before the real deal, or you are restricted to be totally offline for your applications.

Prerequisites

1. Installed *CentOS 6* for PXE server
2. At least two bare metal nodes to work with

High Level Design

1. Manage the tftp directory
   1. /tftpboot/(coreos)(centos)(RHEL)
   2. /tftpboot/pxelinux.0/(MAC) -> linked to Linux image config file

2. Update per install the link for pxelinux
3. Update the DHCP config to reflect the host needing deployment
4. Setup nodes to deploy CoreOS creating an etcd cluster.
5. Have no access to the public etcd discovery tool.
6. Installing the CoreOS slaves to become Kubernetes nodes.

This Guides variables

Node Description	MAC	IP
CoreOS/etcd/Kubernetes Master	d0:00:67:13:0d:00	10.20.30.40
CoreOS Slave 1	d0:00:67:13:0d:01	10.20.30.41
CoreOS Slave 2	d0:00:67:13:0d:02	10.20.30.42

Setup PXELINUX CentOS

To setup CentOS PXELINUX environment there is a complete guide here. This section is the abbreviated version.

1. Install packages needed on CentOS

```
sudo yum install tftp-server dhcp syslinux
```

2. `vi /etc/xinetd.d/tftp` to enable tftp service and change disable to 'no'

```
-disable = no
```

3. Copy over the syslinux images we will need.

```
su -
mkdir -p /tftpboot
cd /tftpboot
cp /usr/share/syslinux/pxelinux.0 /tftpboot
cp /usr/share/syslinux/menu.c32 /tftpboot
cp /usr/share/syslinux/memdisk /tftpboot
cp /usr/share/syslinux/mboot.c32 /tftpboot
cp /usr/share/syslinux/chain.c32 /tftpboot

/sbin/service dhcpd start
/sbin/service xinetd start
/sbin/chkconfig tftp on
```

4. Setup default boot menu

```
mkdir /tftpboot/pxelinux.cfg
touch /tftpboot/pxelinux.cfg/default
```

5. Edit the menu `vi /tftpboot/pxelinux.cfg/default`

```
default menu.c32
prompt 0
timeout 15
ONTIMEOUT local
display boot.msg

MENU TITLE Main Menu

LABEL local
        MENU LABEL Boot local hard drive
        LOCALBOOT 0
```

Now you should have a working PXELINUX setup to image CoreOS nodes. You can verify the services by using VirtualBox locally or with bare metal servers.

Adding CoreOS to PXE

This section describes how to setup the CoreOS images to live alongside a pre-existing PXELINUX environment. 1. Find or create the TFTP root directory that everything will be based on. - For this document we will assume `/tftpboot/` is our root directory.
2. Once we know and have our tftp root directory we will create a new directory structure for our CoreOS images.
3. Download the CoreOS PXE files provided by the CoreOS team.

```
MY_TFTPROOT_DIR=/tftpboot
mkdir -p $MY_TFTPROOT_DIR/images/coreos/
cd $MY_TFTPROOT_DIR/images/coreos/
wget http://stable.release.core-os.net/amd64-usr/current/coreos_production_pxe
wget http://stable.release.core-os.net/amd64-usr/current/coreos_production_pxe
wget http://stable.release.core-os.net/amd64-usr/current/coreos_production_pxe
wget http://stable.release.core-os.net/amd64-usr/current/coreos_production_pxe
gpg --verify coreos_production_pxe.vmlinuz.sig
gpg --verify coreos_production_pxe_image.cpio.gz.sig
```

1. Edit the menu `vi /tftpboot/pxelinux.cfg/default` again

```
default menu.c32
prompt 0
timeout 300
ONTIMEOUT local
display boot.msg

MENU TITLE Main Menu

LABEL local
        MENU LABEL Boot local hard drive
        LOCALBOOT 0

MENU BEGIN CoreOS Menu

    LABEL coreos-master
        MENU LABEL CoreOS Master
        KERNEL images/coreos/coreos_production_pxe.vmlinuz
        APPEND initrd=images/coreos/coreos_production_pxe_image.cpio.gz cloud

    LABEL coreos-slave
        MENU LABEL CoreOS Slave
        KERNEL images/coreos/coreos_production_pxe.vmlinuz
        APPEND initrd=images/coreos/coreos_production_pxe_image.cpio.gz cloud
MENU END
```

This configuration file will now boot from local drive but have the option to PXE image CoreOS.

DHCP configuration

This section covers configuring the DHCP server to hand out our new images. In this case we are assuming that there are other servers that will boot alongside other images. 1. Add the `filename` to the *host* or *subnet* sections.

```
    filename "/tftpboot/pxelinux.0";
```

1. At this point we want to make pxelinux configuration files that will be the templates for the different CoreOS deployments.

```
subnet 10.20.30.0 netmask 255.255.255.0 {
        next-server 10.20.30.242;
        option broadcast-address 10.20.30.255;
        filename "<other default image>";

        ...
        # http://www.syslinux.org/wiki/index.php/PXELINUX
        host core_os_master {
                hardware ethernet d0:00:67:13:0d:00;
                option routers 10.20.30.1;
                fixed-address 10.20.30.40;
                option domain-name-servers 10.20.30.242;
                filename "/pxelinux.0";
        }
        host core_os_slave {
                hardware ethernet d0:00:67:13:0d:01;
                option routers 10.20.30.1;
                fixed-address 10.20.30.41;
                option domain-name-servers 10.20.30.242;
                filename "/pxelinux.0";
        }
        host core_os_slave2 {
                hardware ethernet d0:00:67:13:0d:02;
                option routers 10.20.30.1;
                fixed-address 10.20.30.42;
                option domain-name-servers 10.20.30.242;
                filename "/pxelinux.0";
        }
        ...
}
```

We will be specifying the node configuration later in the guide.

Kubernetes

To deploy our configuration we need to create an `etcd` master. To do so we want to pxe CoreOS with a specific cloud-config.yml. There are two options we have here. 1. Is to template the cloud config file and programmatically create new static configs for different cluster setups. 2. Have a service discovery protocol running in our stack to do auto discovery.

This demo we just make a static single `etcd` server to host our Kubernetes and `etcd` master servers.

Since we are OFFLINE here most of the helping processes in CoreOS and Kubernetes are then limited. To do our setup we will then have to download and serve up our binaries for Kubernetes in our local environment.

An easy solution is to host a small web server on the DHCP/TFTP host for all our binaries to make them available to the local CoreOS PXE machines.

To get this up and running we are going to setup a simple `apache` server to serve our binaries needed to bootstrap Kubernetes.

This is on the PXE server from the previous section:

```
rm /etc/httpd/conf.d/welcome.conf
cd /var/www/html/
wget -O kube-register  https://github.com/kelseyhightower/kube-register/releases/d
wget -O setup-network-environment https://github.com/kelseyhightower/setup-network
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget https://storage.googleapis.com/kubernetes-release/release/v0.15.0/bin/linux/a
wget -O flanneld https://storage.googleapis.com/k8s/flanneld
```

This sets up our binaries we need to run Kubernetes. This would need to be enhanced to download from the Internet for updates in the future.

Now for the good stuff!

Cloud Configs

The following config files are tailored for the OFFLINE version of a Kubernetes deployment.

These are based on the work found here: master.yml, node.yml

To make the setup work, you need to replace a few placeholders:

- Replace `<PXE_SERVER_IP>` with your PXE server ip address (e.g. 10.20.30.242)
- Replace `<MASTER_SERVER_IP>` with the Kubernetes master ip address (e.g. 10.20.30.40)
- If you run a private docker registry, replace `rdocker.example.com` with your docker registry dns name.
- If you use a proxy, replace `rproxy.example.com` with your proxy server (and port)
- Add your own SSH public key(s) to the cloud config at the end

master.yml

On the PXE server make and fill in the variables `vi /var/www/html/coreos/pxe-cloud-config-master.yml`.

```
#cloud-config
---
write_files:
  - path: /opt/bin/waiter.sh
    owner: root
    content: |
      #! /usr/bin/bash
      until curl http://127.0.0.1:4001/v2/machines; do sleep 2; done
  - path: /opt/bin/kubernetes-download.sh
    owner: root
    permissions: 0755
    content: |
      #! /usr/bin/bash
      /usr/bin/wget -N -P "/opt/bin" "http://<PXE_SERVER_IP>/kubectl"
      /usr/bin/wget -N -P "/opt/bin" "http://<PXE_SERVER_IP>/kubernetes"
      /usr/bin/wget -N -P "/opt/bin" "http://<PXE_SERVER_IP>/kubecfg"
      chmod +x /opt/bin/*
  - path: /etc/profile.d/opt-path.sh
    owner: root
    permissions: 0755
```

```
      content: |
        #! /usr/bin/bash
        PATH=$PATH/opt/bin
  coreos:
    units:
      - name: 10-eno1.network
        runtime: true
        content: |
          [Match]
          Name=eno1
          [Network]
          DHCP=yes
      - name: 20-nodhcp.network
        runtime: true
        content: |
          [Match]
          Name=en*
          [Network]
          DHCP=none
      - name: get-kube-tools.service
        runtime: true
        command: start
        content: |
          [Service]
          ExecStartPre=-/usr/bin/mkdir -p /opt/bin
          ExecStart=/opt/bin/kubernetes-download.sh
          RemainAfterExit=yes
          Type=oneshot
      - name: setup-network-environment.service
        command: start
        content: |
          [Unit]
          Description=Setup Network Environment
          Documentation=https://github.com/kelseyhightower/setup-network-environment
          Requires=network-online.target
          After=network-online.target
          [Service]
          ExecStartPre=-/usr/bin/mkdir -p /opt/bin
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/setup-net
          ExecStartPre=/usr/bin/chmod +x /opt/bin/setup-network-environment
          ExecStart=/opt/bin/setup-network-environment
          RemainAfterExit=yes
          Type=oneshot
      - name: etcd.service
        command: start
        content: |
          [Unit]
          Description=etcd
          Requires=setup-network-environment.service
          After=setup-network-environment.service
          [Service]
          EnvironmentFile=/etc/network-environment
          User=etcd
          PermissionsStartOnly=true
          ExecStart=/usr/bin/etcd \
          --name ${DEFAULT_IPV4} \
          --addr ${DEFAULT_IPV4}:4001 \
          --bind-addr 0.0.0.0 \
          --cluster-active-size 1 \
          --data-dir /var/lib/etcd \
          --http-read-timeout 86400 \
          --peer-addr ${DEFAULT_IPV4}:7001 \
          --snapshot true
          Restart=always
          RestartSec=10s
      - name: fleet.socket
        command: start
        content: |
          [Socket]
          ListenStream=/var/run/fleet.sock
      - name: fleet.service
        command: start
        content: |
          [Unit]
          Description=fleet daemon
          Wants=etcd.service
          After=etcd.service
          Wants=fleet.socket
          After=fleet.socket
          [Service]
          Environment="FLEET_ETCD_SERVERS=http://127.0.0.1:4001"
          Environment="FLEET_METADATA=role=master"
          ExecStart=/usr/bin/fleetd
          Restart=always
          RestartSec=10s
      - name: etcd-waiter.service
        command: start
        content: |
          [Unit]
          Description=etcd waiter
          Wants=network-online.target
          Wants=etcd.service
          After=etcd.service
          After=network-online.target
          Before=flannel.service
          Before=setup-network-environment.service
          [Service]
          ExecStartPre=/usr/bin/chmod +x /opt/bin/waiter.sh
          ExecStart=/usr/bin/bash /opt/bin/waiter.sh
          RemainAfterExit=true
          Type=oneshot
      - name: flannel.service
        command: start
```

```
        content: |
          [Unit]
          Wants=etcd-waiter.service
          After=etcd-waiter.service
          Requires=etcd.service
          After=etcd.service
          After=network-online.target
          Wants=network-online.target
          Description=flannel is an etcd backed overlay network for containers
          [Service]
          Type=notify
          ExecStartPre=-/usr/bin/mkdir -p /opt/bin
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/flanneld
          ExecStartPre=/usr/bin/chmod +x /opt/bin/flanneld
          ExecStartPre=-/usr/bin/etcdctl mk /coreos.com/network/config '{"Network":"
          ExecStart=/opt/bin/flanneld
      - name: kube-apiserver.service
        command: start
        content: |
          [Unit]
          Description=Kubernetes API Server
          Documentation=https://github.com/kubernetes/kubernetes
          Requires=etcd.service
          After=etcd.service
          [Service]
          ExecStartPre=-/usr/bin/mkdir -p /opt/bin
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kube-apis
          ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-apiserver
          ExecStart=/opt/bin/kube-apiserver \
          --address=0.0.0.0 \
          --port=8080 \
          --service-cluster-ip-range=10.100.0.0/16 \
          --etcd-servers=http://127.0.0.1:4001 \
          --logtostderr=true
          Restart=always
          RestartSec=10
      - name: kube-controller-manager.service
        command: start
        content: |
          [Unit]
          Description=Kubernetes Controller Manager
          Documentation=https://github.com/kubernetes/kubernetes
          Requires=kube-apiserver.service
          After=kube-apiserver.service
          [Service]
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kube-cont
          ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-controller-manager
          ExecStart=/opt/bin/kube-controller-manager \
          --master=127.0.0.1:8080 \
          --logtostderr=true
          Restart=always
          RestartSec=10
      - name: kube-scheduler.service
        command: start
        content: |
          [Unit]
          Description=Kubernetes Scheduler
          Documentation=https://github.com/kubernetes/kubernetes
          Requires=kube-apiserver.service
          After=kube-apiserver.service
          [Service]
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kube-sche
          ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-scheduler
          ExecStart=/opt/bin/kube-scheduler --master=127.0.0.1:8080
          Restart=always
          RestartSec=10
      - name: kube-register.service
        command: start
        content: |
          [Unit]
          Description=Kubernetes Registration Service
          Documentation=https://github.com/kelseyhightower/kube-register
          Requires=kube-apiserver.service
          After=kube-apiserver.service
          Requires=fleet.service
          After=fleet.service
          [Service]
          ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kube-regi
          ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-register
          ExecStart=/opt/bin/kube-register \
          --metadata=role=node \
          --fleet-endpoint=unix:///var/run/fleet.sock \
          --healthz-port=10248 \
          --api-endpoint=http://127.0.0.1:8080
          Restart=always
          RestartSec=10
    update:
      group: stable
      reboot-strategy: off
  ssh_authorized_keys:
    - ssh-rsa AAAAB3NzaC1yc2EAAAAD...
```

node.yml

On the PXE server make and fill in the variables `vi /var/www/html/coreos/pxe-cloud-config-slave.yml`.

```
#cloud-config
---
```

```
write_files:
  - path: /etc/default/docker
    content: |
      DOCKER_EXTRA_OPTS='--insecure-registry="rdocker.example.com:5000"'
coreos:
  units:
    - name: 10-eno1.network
      runtime: true
      content: |
        [Match]
        Name=eno1
        [Network]
        DHCP=yes
    - name: 20-nodhcp.network
      runtime: true
      content: |
        [Match]
        Name=en*
        [Network]
        DHCP=none
    - name: etcd.service
      mask: true
    - name: docker.service
      drop-ins:
        - name: 50-insecure-registry.conf
          content: |
            [Service]
            Environment="HTTP_PROXY=http://rproxy.example.com:3128/" "NO_PROXY=loc
    - name: fleet.service
      command: start
      content: |
        [Unit]
        Description=fleet daemon
        Wants=fleet.socket
        After=fleet.socket
        [Service]
        Environment="FLEET_ETCD_SERVERS=http://<MASTER_SERVER_IP>:4001"
        Environment="FLEET_METADATA=role=node"
        ExecStart=/usr/bin/fleetd
        Restart=always
        RestartSec=10s
    - name: flannel.service
      command: start
      content: |
        [Unit]
        After=network-online.target
        Wants=network-online.target
        Description=flannel is an etcd backed overlay network for containers
        [Service]
        Type=notify
        ExecStartPre=-/usr/bin/mkdir -p /opt/bin
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/flanneld
        ExecStartPre=/usr/bin/chmod +x /opt/bin/flanneld
        ExecStart=/opt/bin/flanneld -etcd-endpoints http://<MASTER_SERVER_IP>:4001
    - name: docker.service
      command: start
      content: |
        [Unit]
        After=flannel.service
        Wants=flannel.service
        Description=Docker Application Container Engine
        Documentation=http://docs.docker.io
        [Service]
        EnvironmentFile=-/etc/default/docker
        EnvironmentFile=/run/flannel/subnet.env
        ExecStartPre=/bin/mount --make-rprivate /
        ExecStart=/usr/bin/docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
        [Install]
        WantedBy=multi-user.target
    - name: setup-network-environment.service
      command: start
      content: |
        [Unit]
        Description=Setup Network Environment
        Documentation=https://github.com/kelseyhightower/setup-network-environment
        Requires=network-online.target
        After=network-online.target
        [Service]
        ExecStartPre=-/usr/bin/mkdir -p /opt/bin
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/setup-net
        ExecStartPre=/usr/bin/chmod +x /opt/bin/setup-network-environment
        ExecStart=/opt/bin/setup-network-environment
        RemainAfterExit=yes
        Type=oneshot
    - name: kube-proxy.service
      command: start
      content: |
        [Unit]
        Description=Kubernetes Proxy
        Documentation=https://github.com/kubernetes/kubernetes
        Requires=setup-network-environment.service
        After=setup-network-environment.service
        [Service]
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kube-prox
        ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-proxy
        ExecStart=/opt/bin/kube-proxy \
        --etcd-servers=http://<MASTER_SERVER_IP>:4001 \
        --logtostderr=true
        Restart=always
        RestartSec=10
    - name: kube-kubelet.service
      command: start
      content: |
        [Unit]
        Description=Kubernetes Kubelet
        Documentation=https://github.com/kubernetes/kubernetes
        Requires=setup-network-environment.service
        After=setup-network-environment.service
        [Service]
        EnvironmentFile=/etc/network-environment
        ExecStartPre=/usr/bin/wget -N -P /opt/bin http://<PXE_SERVER_IP>/kubelet
        ExecStartPre=/usr/bin/chmod +x /opt/bin/kubelet
        ExecStart=/opt/bin/kubelet \
        --address=0.0.0.0 \
        --port=10250 \
        --hostname-override=${DEFAULT_IPV4} \
        --api-servers=<MASTER_SERVER_IP>:8080 \
        --healthz-bind-address=0.0.0.0 \
        --healthz-port=10248 \
        --logtostderr=true
        Restart=always
        RestartSec=10
  update:
    group: stable
    reboot-strategy: off
ssh_authorized_keys:
  - ssh-rsa AAAAB3NzaC1yc2EAAAAD...
```

New pxelinux.cfg file

Create a pxelinux target file for a *slave* node: `vi /tftpboot/pxelinux.cfg/coreos-node-slave`

```
default coreos
prompt 1
timeout 15

display boot.msg

label coreos
    menu default
    kernel images/coreos/coreos_production_pxe.vmlinuz
    append initrd=images/coreos/coreos_production_pxe_image.cpio.gz cloud-config-u
```

And one for the *master* node: `vi /tftpboot/pxelinux.cfg/coreos-node-master`

```
default coreos
prompt 1
timeout 15

display boot.msg

label coreos
    menu default
    kernel images/coreos/coreos_production_pxe.vmlinuz
    append initrd=images/coreos/coreos_production_pxe_image.cpio.gz cloud-config-u
```

Specify the pxelinux targets

Now that we have our new targets setup for master and slave we want to configure the specific hosts to those targets. We will do this by using the pxelinux mechanism of setting a specific MAC addresses to a specific pxelinux.cfg file.

Refer to the MAC address table in the beginning of this guide. Documentation for more details can be found here.

```
cd /tftpboot/pxelinux.cfg
ln -s coreos-node-master 01-d0-00-67-13-0d-00
ln -s coreos-node-slave 01-d0-00-67-13-0d-01
ln -s coreos-node-slave 01-d0-00-67-13-0d-02
```

Reboot these servers to get the images PXEd and ready for running containers!

Creating test pod

Now that the CoreOS with Kubernetes installed is up and running lets spin up some Kubernetes pods to demonstrate the system.

See a simple nginx example to try out your new cluster.

For more complete applications, please look in the examples directory.

Helping commands for debugging

List all keys in etcd:

```
etcdctl ls --recursive
```

List fleet machines

```
fleetctl list-machines
```

Check system status of services on master:

```
systemctl status kube-apiserver
systemctl status kube-controller-manager
systemctl status kube-scheduler
systemctl status kube-register
```

Check system status of services on a node:

```
systemctl status kube-kubelet
systemctl status docker.service
```

List Kubernetes

```
kubectl get pods
kubectl get nodes
```

Kill all pods:

```
for i in `kubectl get pods | awk '{print $1}'`; do kubectl delete pod $i; done
```

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Bare-metal (Offline)	CoreOS	CoreOS	flannel	docs		Community (@jeffbean)

For support level information on all solutions, see the Table of solutions chart.

Fedora via Ansible

Configuring Kubernetes on Fedora via Ansible offers a simple way to quickly create a clustered environment with little effort.

Prerequisites

1. Host able to run ansible and able to clone the following repo: Kubernetes
2. A Fedora 21+ host to act as cluster master
3. As many Fedora 21+ hosts as you would like, that act as cluster nodes

The hosts can be virtual or bare metal. Ansible will take care of the rest of the configuration for you - configuring networking, installing packages, handling the firewall, etc. This example will use one master and two nodes.

Architecture of the cluster

A Kubernetes cluster requires etcd, a master, and n nodes, so we will create a cluster with three hosts, for example:

```
master,etcd = kube-master.example.com
    node1 = kube-node-01.example.com
    node2 = kube-node-02.example.com
```

Make sure your local machine has

- ansible (must be 1.9.0+)
- git
- python-netaddr

If not

```
dnf install -y ansible git python-netaddr
```

Now clone down the Kubernetes repository

```
git clone https://github.com/kubernetes/contrib.git
cd contrib/ansible
```

Tell ansible about each machine and its role in your cluster

Get the IP addresses from the master and nodes. Add those to the `~/contrib/ansible/inventory/localhost.ini` file on the host running Ansible.

```
[masters]
kube-master.example.com

[etcd]
kube-master.example.com

[nodes]
kube-node-01.example.com
kube-node-02.example.com
```

Setting up ansible access to your nodes

If you already are running on a machine which has passwordless ssh access to the kube-master and kube-node-{01,02} nodes, and 'sudo' privileges, simply set the value of `ansible_ssh_user` in `~/contrib/ansible/inventory/group_vars/all.yml` to the username which you use to ssh to the nodes (i.e. `fedora`), and proceed to the next step...

Otherwise setup ssh on the machines like so (you will need to know the root password to all machines in the cluster).

edit: `~/contrib/ansible/inventory/group_vars/all.yml`

```
ansible_ssh_user: root
```

Configuring ssh access to the cluster

If you already have ssh access to every machine using ssh public keys you may skip to setting up the cluster

Make sure your local machine (root) has an ssh key pair if not

```
ssh-keygen
```

Copy the ssh public key to **all** nodes in the cluster

```
for node in kube-master.example.com kube-node-01.example.com kube-node-02.example.
  ssh-copy-id ${node}
done
```

Setting up the cluster

Although the default value of variables in `~/contrib/ansible/inventory/group_vars/all.yml` should be good enough, if not, change them as needed.

```
edit: ~/contrib/ansible/inventory/group_vars/all.yml
```

Configure access to Kubernetes packages

Modify `source_type` as below to access Kubernetes packages through the package manager.

```
source_type: packageManager
```

Configure the IP addresses used for services

Each Kubernetes service gets its own IP address. These are not real IPs. You need to only select a range of IPs which are not in use elsewhere in your environment.

```
kube_service_addresses: 10.254.0.0/16
```

Managing flannel

Modify `flannel_subnet`, `flannel_prefix` and `flannel_host_prefix` only if defaults are not appropriate for your cluster.

Managing add on services in your cluster

Set `cluster_logging` to false or true (default) to disable or enable logging with elasticsearch.

```
cluster_logging: true
```

Turn `cluster_monitoring` to true (default) or false to enable or disable cluster monitoring with heapster and influxdb.

```
cluster_monitoring: true
```

Turn `dns_setup` to true (recommended) or false to enable or disable whole DNS configuration.

```
dns_setup: true
```

Tell ansible to get to work!

This will finally setup your whole Kubernetes cluster for you.

```
cd ~/contrib/ansible/scripts/

./deploy-cluster.sh
```

Testing and using your new cluster

That's all there is to it. It's really that easy. At this point you should have a functioning Kubernetes cluster.

Show Kubernetes nodes

Run the following on the kube-master:

```
kubectl get nodes
```

Show services running on masters and nodes

```
systemctl | grep -i kube
```

Show firewall rules on the masters and nodes

```
iptables -nvL
```

Create /tmp/apache.json on the master with the following contents and deploy pod

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "fedoraapache",
    "labels": {
      "name": "fedoraapache"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "fedoraapache",
        "image": "fedora/apache",
        "ports": [
          {
            "hostPort": 80,
            "containerPort": 80
          }
        ]
      }
    ]
  }
}
```

```
kubectl create -f /tmp/apache.json
```

Check where the pod was created

```
kubectl get pods
```

Check Docker status on nodes

```
docker ps
docker images
```

After the pod is 'Running' Check web server access on the node

```
curl http://localhost
```

That's it!

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Bare-metal	Ansible	Fedora	flannel	docs		Project

For support level information on all solutions, see the Table of solutions chart.

Fedora (Single Node)

Prerequisites

1. You need 2 or more machines with Fedora installed. These can be either bare metal machines or virtual machines.

Instructions

This is a getting started guide for Fedora. It is a manual configuration so you understand all the underlying packages / services / ports, etc…

This guide will only get ONE node (previously minion) working. Multiple nodes require a functional networking configuration done outside of Kubernetes. Although the additional Kubernetes configuration requirements should be obvious.

The Kubernetes package provides a few services: kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, kube-proxy. These services are managed by systemd and the configuration resides in a central location: /etc/kubernetes. We will break the services up between the hosts. The first host, fed-master, will be the Kubernetes master. This host will run the kube-apiserver, kube-controller-manager, and kube-scheduler. In addition, the master will also run *etcd* (not needed if *etcd* runs on a different host but this guide assumes that *etcd* and Kubernetes master run on the same host). The remaining host, fed-node will be the node and run kubelet, proxy and docker.

System Information:

Hosts:

```
fed-master = 192.168.121.9
fed-node = 192.168.121.65
```

Prepare the hosts:

- Install Kubernetes on all hosts - fed-{master,node}. This will also pull in docker. Also install etcd on fed-master. This guide has been tested with Kubernetes-0.18 and beyond.
- Running on AWS EC2 with RHEL 7.2, you need to enable "extras" repository for yum by editing `/etc/yum.repos.d/redhat-rhui.repo` and changing the `enable=0` to `enable=1` for extras.

```
dnf -y install kubernetes
```

- Install etcd

```
dnf -y install etcd
```

- Add master and node to /etc/hosts on all machines (not needed if hostnames already in DNS). Make sure that communication works between fed-master and fed-node by using a utility such as ping.

```
echo "192.168.121.9    fed-master
192.168.121.65    fed-node" >> /etc/hosts
```

- Edit /etc/kubernetes/config (which should be the same on all hosts) to set the name of the master server:

```
# Comma separated list of nodes in the etcd cluster
KUBE_MASTER="--master=http://fed-master:8080"
```

- Disable the firewall on both the master and node, as docker does not play well with other firewall rule managers. Please note that iptables-services does not exist on default fedora server install.

```
systemctl disable iptables-services firewalld
systemctl stop iptables-services firewalld
```

Configure the Kubernetes services on the master.

- Edit /etc/kubernetes/apiserver to appear as such. The service-cluster-ip-range IP addresses must be an unused block of addresses, not used anywhere else. They do not need to be routed or assigned to anything.

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=http://127.0.0.1:2379"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

# Add your own!
KUBE_API_ARGS=""
```

- Edit /etc/etcd/etcd.conf to let etcd listen on all available IPs instead of 127.0.0.1. If you have not done this, you might see an error such as “connection refused”.

```
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
```

- Start the appropriate services on master:

```
for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

- Addition of nodes:

- Create following node.json file on Kubernetes master node:

```
{
    "apiVersion": "v1",
    "kind": "Node",
    "metadata": {
        "name": "fed-node",
        "labels":{ "name": "fed-node-label"}
    },
    "spec": {
        "externalID": "fed-node"
    }
}
```

Now create a node object internally in your Kubernetes cluster by running:

```
$ kubectl create -f ./node.json

$ kubectl get nodes
NAME            STATUS        AGE      VERSION
fed-node        Unknown       4h
```

Please note that in the above, it only creates a representation for the node *fed-node* internally. It does not provision the actual *fed-node*. Also, it is assumed that *fed-node* (as specified in `name`) can be resolved and is reachable from Kubernetes master node. This guide will discuss how to provision a Kubernetes node (fed-node) below.

Configure the Kubernetes services on the node.

We need to configure the kubelet on the node.

- Edit /etc/kubernetes/kubelet to appear as such:

```
###
# Kubernetes kubelet (node) config

# The address for the info server to serve on (set to 0.0.0.0 or "" for all interf
KUBELET_ADDRESS="--address=0.0.0.0"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname-override=fed-node"

# location of the api-server
KUBELET_API_SERVER="--api-servers=http://fed-master:8080"

# Add your own!
#KUBELET_ARGS=""
```

- Start the appropriate services on the node (fed-node).

```
for SERVICES in kube-proxy kubelet docker; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

- Check to make sure now the cluster can see the fed-node on fed-master, and its status changes to *Ready*.

```
kubectl get nodes
NAME            STATUS      AGE       VERSION
fed-node        Ready       4h
```

- Deletion of nodes:

To delete *fed-node* from your Kubernetes cluster, one should run the following on fed-master (Please do not do it, it is just for information):

```
kubectl delete -f ./node.json
```

You should be finished!

The cluster should be running! Launch a test pod.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Bare-metal	custom	Fedora	*none*	docs		Project

For support level information on all solutions, see the Table of solutions chart.

Fedora (Multi Node)

This document describes how to deploy Kubernetes on multiple hosts to set up a multi-node cluster and networking with flannel. Follow fedora getting started guide to setup 1 master (fed-master) and 2 or more nodes. Make sure that all nodes have different names (fed-node1, fed-node2 and so on) and labels (fed-node1-label, fed-node2-label, and so on) to avoid any conflict. Also make sure that the Kubernetes master host is running etcd, kube-controller-manager, kube-scheduler, and kube-apiserver services, and the nodes are running docker, kube-proxy and kubelet services. Now install flannel on Kubernetes nodes. Flannel on each node configures an overlay network that docker uses. Flannel runs on each node to setup a unique class-C container network.

Prerequisites

You need 2 or more machines with Fedora installed.

Master Setup

Perform following commands on the Kubernetes master

- Configure flannel by creating a `flannel-config.json` in your current directory on fed-master. Flannel provides udp and vxlan among other overlay networking backend options. In this guide, we choose kernel based vxlan backend. The contents of the json are:

```
{
    "Network": "18.16.0.0/16",
    "SubnetLen": 24,
    "Backend": {
        "Type": "vxlan",
        "VNI": 1
     }
}
```

NOTE: Choose an IP range that is *NOT* part of the public IP address range.

Add the configuration to the etcd server on fed-master.

```
etcdctl set /coreos.com/network/config < flannel-config.json
```

- Verify that the key exists in the etcd server on fed-master.

```
etcdctl get /coreos.com/network/config
```

Node Setup

Perform following commands on all Kubernetes nodes

Install the flannel package

```
# dnf -y install flannel
```

Edit the flannel configuration file /etc/sysconfig/flanneld as follows:

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://fed-master:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS=""
```

Note: By default, flannel uses the interface for the default route. If you have multiple interfaces and would like to use an interface other than the default route one, you could add “-iface=” to FLANNEL_OPTIONS. For additional options, run `flanneld --help` on command line.

Enable the flannel service.

```
systemctl enable flanneld
```

If docker is not running, then starting flannel service is enough and skip the next step.

```
systemctl start flanneld
```

If docker is already running, then stop docker, delete docker bridge (docker0), start flanneld and restart docker as follows. Another alternative is to just reboot the system (`systemctl reboot`).

```
systemctl stop docker
ip link delete docker0
systemctl start flanneld
systemctl start docker
```

Test the cluster and flannel configuration

Now check the interfaces on the nodes. Notice there is now a flannel.1 interface, and the ip addresses of docker0 and flannel.1 interfaces are in the same network. You will notice that docker0 is assigned a subnet (18.16.29.0/24 as shown below) on each Kubernetes node out of the IP range configured above. A working output should look like this:

```
# ip -4 a|grep inet
    inet 127.0.0.1/8 scope host lo
    inet 192.168.122.77/24 brd 192.168.122.255 scope global dynamic eth0
    inet 18.16.29.0/16 scope global flannel.1
    inet 18.16.29.1/24 scope global docker0
```

From any node in the cluster, check the cluster members by issuing a query to etcd server via curl (only partial output is shown using `grep -E "\{|\}|key|value"`). If you set up a 1 master and 3 nodes cluster, you should see one block for each node showing the subnets they have been assigned. You can associate those subnets to each node by the MAC address (VtepMAC) and IP address (Public IP) that is listed in the output.

```
curl -s http://fed-master:2379/v2/keys/coreos.com/network/subnets | python -mjson.
```

```
{
    "node": {
        "key": "/coreos.com/network/subnets",
            {
                "key": "/coreos.com/network/subnets/18.16.29.0-24",
                "value": "{\"PublicIP\":\"192.168.122.77\",\"BackendType\":\"vxlan
            },
            {
                "key": "/coreos.com/network/subnets/18.16.83.0-24",
                "value": "{\"PublicIP\":\"192.168.122.36\",\"BackendType\":\"vxlan
            },
            {
                "key": "/coreos.com/network/subnets/18.16.90.0-24",
                "value": "{\"PublicIP\":\"192.168.122.127\",\"BackendType\":\"vxla
            }
    }
}
```

From all nodes, review the `/run/flannel/subnet.env` file. This file was generated automatically by flannel.

```
# cat /run/flannel/subnet.env
FLANNEL_SUBNET=18.16.29.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

At this point, we have etcd running on the Kubernetes master, and flannel / docker running on Kubernetes nodes. Next steps are for testing cross-host container communication which will confirm that docker and flannel are configured properly.

Issue the following commands on any 2 nodes:

```
# docker run -it fedora:latest bash
bash-4.3#
```

This will place you inside the container. Install iproute and iputils packages to install ip and ping utilities. Due to a bug, it is required to modify capabilities of ping binary to work around "Operation not permitted" error.

```
bash-4.3# dnf -y install iproute iputils
bash-4.3# setcap cap_net_raw-ep /usr/bin/ping
```

Now note the IP address on the first node:

```
bash-4.3# ip -4 a l eth0 | grep inet
    inet 18.16.29.4/24 scope global eth0
```

And also note the IP address on the other node:

```
bash-4.3# ip a l eth0 | grep inet
    inet 18.16.90.4/24 scope global eth0
```

Now ping from the first node to the other node:

```
bash-4.3# ping 18.16.90.4
PING 18.16.90.4 (18.16.90.4) 56(84) bytes of data.
64 bytes from 18.16.90.4: icmp_seq=1 ttl=62 time=0.275 ms
64 bytes from 18.16.90.4: icmp_seq=2 ttl=62 time=0.372 ms
```

Now Kubernetes multi-node cluster is set up with overlay networking set up by flannel.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Bare-metal	custom	Fedora	flannel	docs		Community (@aveshagarwal)
libvirt	custom	Fedora	flannel	docs		Community (@aveshagarwal)
KVM	custom	Fedora	flannel	docs		Community (@aveshagarwal)

For support level information on all solutions, see the Table of solutions chart.

CentOS

Warning

This guide has been deprecated. It was originally written for Kubernetes 1.1.0. Please check the latest guide.

Prerequisites

To configure Kubernetes with CentOS, you'll need a machine to act as a master, and one or more CentOS 7 hosts to act as cluster nodes.

Starting a cluster

This is a getting started guide for CentOS. It is a manual configuration so you understand all the underlying packages / services / ports, etc…

The Kubernetes package provides a few services: kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, kube-proxy. These services are managed by systemd and the configuration resides in a central location: /etc/kubernetes. We will break the services up between the hosts. The first host, centos-master, will be the Kubernetes master. This host will run the kube-apiserver, kube-controller-manager and kube-scheduler. In addition, the master will also run *etcd*. The remaining hosts, centos-minion-n will be the nodes and run kubelet, proxy, cadvisor and docker.

All of them run flanneld as networking overlay.

System Information:

Hosts:

Please replace host IP with your environment.

```
centos-master = 192.168.121.9
centos-minion-1 = 192.168.121.65
centos-minion-2 = 192.168.121.66
centos-minion-3 = 192.168.121.67
```

Prepare the hosts:

- Create a /etc/yum.repos.d/virt7-docker-common-release.repo on all hosts - centos-{master,minion-n} with following information.

```
[virt7-docker-common-release]
name=virt7-docker-common-release
baseurl=http://cbs.centos.org/repos/virt7-docker-common-release/x86_64/os/
gpgcheck=0
```

- Install Kubernetes, etcd and flannel on all hosts - centos-{master,minion-n}. This will also pull in docker and cadvisor.

```
yum -y install --enablerepo=virt7-docker-common-release kubernetes etcd flannel
```

- Add master and node to /etc/hosts on all machines (not needed if hostnames already in DNS)

```
echo "192.168.121.9    centos-master
192.168.121.65    centos-minion-1
192.168.121.66  centos-minion-2
192.168.121.67  centos-minion-3" >> /etc/hosts
```

- Edit /etc/kubernetes/config which will be the same on all hosts to contain:

```
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow-privileged=false"

# How the replication controller and scheduler find the kube-apiserver
KUBE_MASTER="--master=http://centos-master:8080"
```

- Disable the firewall on the master and all the nodes, as docker does not play well with other firewall rule managers. CentOS won't let you disable the firewall as long as SELinux is enforcing, so that needs to be disabled first.

```
setenforce 0
systemctl disable iptables-services firewalld
systemctl stop iptables-services firewalld
```

Configure the Kubernetes services on the master.

- Edit /etc/etcd/etcd.conf to appear as such:

```
# [member]
ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"

#[cluster]
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

- Edit /etc/kubernetes/apiserver to appear as such:

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"

# The port on the local server to listen on.
KUBE_API_PORT="--port=8080"

# Port kubelets listen on
KUBELET_PORT="--kubelet-port=10250"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=http://centos-master:2379"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

# Add your own!
KUBE_API_ARGS=""
```

- Start ETCD and configure it to hold the network overlay configuration on master: **Warning** This network must be unused in your network infrastructure! `172.30.0.0/16` is free in our network.

```
systemctl start etcd
etcdctl mkdir /kube-centos/network
etcdctl mk /kube-centos/network/config "{ \"Network\": \"172.30.0.0/16\", \"Subnet
```

- Configure flannel to overlay Docker network in /etc/sysconfig/flanneld on the master (also in the nodes as we'll see):

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD_ENDPOINTS="http://centos-master:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/kube-centos/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

- Start the appropriate services on master:

```
for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler flannel
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

Configure the Kubernetes services on the nodes.

We need to configure the kubelet and start the kubelet and proxy

- Edit /etc/kubernetes/kubelet to appear as such:

```
# The address for the info server to serve on
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
# Check the node number!
KUBELET_HOSTNAME="--hostname-override=centos-minion-n"

# Location of the api-server
KUBELET_API_SERVER="--api-servers=http://centos-master:8080"

# Add your own!
KUBELET_ARGS=""
```

- Configure flannel to overlay Docker network in /etc/sysconfig/flanneld (in all the nodes)

```
# Flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD_ENDPOINTS="http://centos-master:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_PREFIX="/kube-centos/network"

# Any additional options that you want to pass
#FLANNEL_OPTIONS=""
```

- Start the appropriate services on node (centos-minion-n).

```
for SERVICES in kube-proxy kubelet flanneld docker; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

- Configure kubectl

```
kubectl config set-cluster default-cluster --server=http://centos-master:8080
kubectl config set-context default-context --cluster=default-cluster --user=defaul
kubectl config use-context default-context
```

You should be finished!

- Check to make sure the cluster can see the node (on centos-master)

```
$ kubectl get nodes
NAME                   STATUS     AGE     VERSION
centos-minion-1        Ready      3d      v1.6.0+fff5156
centos-minion-2        Ready      3d      v1.6.0+fff5156
centos-minion-3        Ready      3d      v1.6.0+fff5156
```

The cluster should be running! Launch a test pod.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Bare-metal	custom	CentOS	flannel	docs		Community (@coolsvap)

For support level information on all solutions, see the Table of solutions chart.

CoreOS on AWS or GCE

- Official CoreOS Guides
- Community Guides
- Support Level

There are multiple guides on running Kubernetes with CoreOS:

Official CoreOS Guides

These guides are maintained by CoreOS and deploy Kubernetes the "CoreOS Way" with full TLS, the DNS add-on, and more. These guides pass Kubernetes conformance testing and we encourage you to test this yourself.

AWS Multi-Node

Guide and CLI tool for setting up a multi-node cluster on AWS. CloudFormation is used to set up a master and multiple workers in auto-scaling groups.

Bare Metal Multi-Node

Guide and HTTP/API service for PXE booting and provisioning a multi-node cluster on bare metal. Ignition is used to provision a master and multiple workers on the first boot from disk.

Vagrant Multi-Node

Guide to setting up a multi-node cluster on Vagrant. The deployer can independently configure the number of etcd nodes, master nodes, and worker nodes to bring up a fully HA control plane.

Vagrant Single-Node

The quickest way to set up a Kubernetes development environment locally. As easy as `git clone`, `vagrant up` and configuring `kubectl`.

Full Step by Step Guide

A generic guide to setting up an HA cluster on any cloud or bare metal, with full TLS. Repeat the master or worker steps to configure more machines of that role.

Community Guides

These guides are maintained by community members, cover specific platforms and use cases, and experiment with different ways of configuring Kubernetes on CoreOS.

Easy Multi-node Cluster on Google Compute Engine

Scripted installation of a single master, multi-worker cluster on GCE. Kubernetes components are managed by fleet.

Multi-node cluster using cloud-config and Weave on Vagrant

Configure a Vagrant-based cluster of 3 machines with networking provided by Weave.

Multi-node cluster using cloud-config and Vagrant

Configure a single master, multi-worker cluster locally, running on your choice of hypervisor: VirtualBox, Parallels, or VMware

Single-node cluster using a small OS X App

Guide to running a solo cluster (master + worker) controlled by an OS X menubar application. Uses xhyve + CoreOS under the hood.

Multi-node cluster with Vagrant and fleet units using a small OS X App

Guide to running a single master, multi-worker cluster controlled by an OS X menubar application. Uses Vagrant under the hood.

Multi-node cluster using cloud-config, CoreOS and VMware ESXi

Configure a single master, single worker cluster on VMware ESXi.

Single/Multi-node cluster using cloud-config, CoreOS and Foreman

Configure a standalone Kubernetes or a Kubernetes cluster with Foreman.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
GCE	CoreOS	CoreOS	flannel	docs		Community (@pires)
Vagrant	CoreOS	CoreOS	flannel	docs		Community (@pires, @AntonioMeireles)

For support level information on all solutions, see the Table of solutions chart.

Kubernetes on Ubuntu

There are multiple ways to run a Kubernetes cluster with Ubuntu. These pages explain how to deploy Kubernetes on Ubuntu on multiple public and private clouds, as well as bare metal.

Official Ubuntu Guides

- The Canonical Distribution of Kubernetes

Supports AWS, GCE, Azure, Joyent, OpenStack, VMWare, Bare Metal and localhost deployments.

Quick Start

conjure-up provides the quickest way to deploy Kubernetes on Ubuntu for multiple clouds and bare metal. It provides a user-friendly UI that prompts you for cloud credentials and configuration options

Available for Ubuntu 16.04 and newer:

```
sudo snap install conjure-up --classic
# re-login may be required at that point if you just installed snap utility
conjure-up kubernetes
```

As well as Homebrew for macOS:

```
brew install conjure-up
conjure-up kubernetes
```

Operational Guides

These are more in-depth guides for users choosing to run Kubernetes in production:

- Installation
- Validation
- Backups
- Upgrades
- Scaling
- Logging
- Monitoring
- Networking
- Security
- Storage
- Troubleshooting
- Decommissioning
- Operational Considerations
- Glossary

Developer Guides

- Localhost using LXD

Where to find us

We're normally following the following Slack channels:

- sig-cluster-lifecycle
- sig-cluster-ops

- sig-onprem

and we monitor the Kubernetes mailing lists.

Validation

This page will outline how to ensure that a Juju deployed Kubernetes cluster has stood up correctly and is ready to accept workloads.

Before you begin

This page assumes you have a working Juju deployed cluster.

Validation

End to End Testing

End-to-end (e2e) tests for Kubernetes provide a mechanism to test end-to-end behavior of the system, and is the last signal to ensure end user operations match developer specifications. Although unit and integration tests provide a good signal, in a distributed system like Kubernetes it is not uncommon that a minor change may pass all unit and integration tests, but cause unforeseen changes at the system level.

The primary objectives of the e2e tests are to ensure a consistent and reliable behavior of the kubernetes code base, and to catch hard-to-test bugs before users do, when unit and integration tests are insufficient.

Usage

To deploy the end-to-end test suite, you need to relate the `kubernetes-e2e` charm to your existing kubernetes-master nodes and easyrsa:

```
juju deploy cs:~containers/kubernetes-e2e
juju add-relation kubernetes-e2e kubernetes-master
juju add-relation kubernetes-e2e easyrsa
```

Once the relations have settled, you can do `juju status` until the workload status results in `Ready to test.` - you may then kick off an end to end validation test.

Running the e2e test

The e2e test is encapsulated as an action to ensure consistent runs of the end to end test. The defaults are sensible for most deployments.

```
juju run-action kubernetes-e2e/0 test
```

Tuning the e2e test

The e2e test is configurable. By default it will focus on or skip the declared conformance tests in a cloud agnostic way. Default behaviors are configurable. This allows the operator to test only a subset of the conformance tests, or to test more behaviors not enabled by default. You can see all tunable options on the charm by inspecting the schema output of the actions:

```
juju actions kubernetes-e2e --format=yaml --schema
```

Output:

```
test:
  description: Run end-to-end validation test suite
  properties:
    focus:
      default: \[Conformance\]
      description: Regex focus for executing the test
      type: string
    skip:
      default: \[Flaky\]
      description: Regex of tests to skip
      type: string
    timeout:
      default: 30000
      description: Timeout in nanoseconds
      type: integer
  title: test
  type: object
```

As an example, you can run a more limited set of tests for rapid validation of a deployed cluster. The following example will skip the `Flaky`, `Slow`, and `Feature` labeled tests:

```
juju run-action kubernetes-e2e/0 test skip='\[(Flaky|Slow|Feature:.*)\]'
```

Note: the escaping of the regex due to how bash handles brackets.
To see the different types of tests the Kubernetes end-to-end charm has access to, we encourage you to see the upstream documentation on the different types of tests, and to strongly understand what subsets of the tests you are running.

Kinds of tests

More information on end-to-end testing

Along with the above descriptions, end-to-end testing is a much larger subject than this readme can encapsulate. There is far more information in the end-to-end testing guide.

Evaluating end-to-end results

It is not enough to just simply run the test. Result output is stored in two places. The raw output of the e2e run is available in the `juju show-action-output` command, as well as a flat file on disk on the `kubernetes-e2e` unit that executed the test.

Note: The results will only be available once the action has completed the test run. End-to-end testing can be quite time intensive. Often times taking **greater than 1 hour**, depending on configuration.

Flat file

Here's how to copy the output out as a file:

```
juju run-action kubernetes-e2e/0 test
```

Output:

```
Action queued with id: 4ceed33a-d96d-465a-8f31-20d63442e51b
```

Copy output to your local machine:

```
juju scp kubernetes-e2e/0:4ceed33a-d96d-465a-8f31-20d63442e51b.log .
```

Action result output

Or you can just show the output inline:

```
juju run-action kubernetes-e2e/0 test
```

Output:

```
Action queued with id: 4ceed33a-d96d-465a-8f31-20d63442e51b
```

Show the results in your terminal:

```
juju show-action-output 4ceed33a-d96d-465a-8f31-20d63442e51b
```

Known issues

The e2e test suite assumes egress network access. It will pull container images from `gcr.io`. You will need to have this registry unblocked in your firewall to successfully run e2e test results. Or you may use the exposed proxy settings properly configured on the kubernetes-worker units.

Upgrading the e2e tests

The e2e tests are always expanding, you can see if there's an upgrade available by running `juju status kubernetes-e2e`.

When an upgrade is available, upgrade your deployment:

```
juju upgrade-charm kubernetes-e2e
```

Backups

This page shows you how to backup and restore data from the different deployed services in a given cluster.

Before you begin

This page assumes you have a working Juju deployed cluster.

Exporting cluster data

Exporting of cluster data is not supported at this time.

Restoring cluster data

Importing of cluster data is not supported at this time.

Exporting etcd data

Migrating etcd is a fairly easy task.

Step 1: Snapshot your existing cluster. This is encapsulated in the `snapshot` action.

```
juju run-action etcd/0 snapshot
```

Results:

```
Action queued with id: b46d5d6f-5625-4320-8cda-b611c6ae580c
```

Step 2: Check the status of the action so you can grab the snapshot and verify the sum. The copy.cmd result output is a copy/paste command for you to download the exact snapshot that you just created.

Download the snapshot archive from the unit that created the snapshot and verify the sha256 sum

```
juju show-action-output b46d5d6f-5625-4320-8cda-b611c6ae580c
```

Results:

```
results:
  copy:
    cmd: juju scp etcd/0:/home/ubuntu/etcd-snapshots/etcd-snapshot-2016-11-09-02.4
      .
  snapshot:
    path: /home/ubuntu/etcd-snapshots/etcd-snapshot-2016-11-09-02.41.47.tar.gz
    sha256: 1dea04627812397c51ee87e313433f3102f617a9cab1d1b79698323f6459953d
    size: 68K
status: completed
```

Copy the snapshot to the local disk and then check the sha256sum.

```
juju scp etcd/0:/home/ubuntu/etcd-snapshots/etcd-snapshot-2016-11-09-02.41.47.tar.
sha256sum etcd-snapshot-2016-11-09-02.41.47.tar.gz
```

Step 3: Deploy the new cluster leader, and attach the snapshot:

```
juju deploy etcd new-etcd --resource snapshot=./etcd-snapshot-2016-11-09-02.41.47.
```

Step 4: Re-Initialize the master with the data from the resource we just attached in step 3.

```
juju run-action new-etcd/0 restore
```

Restoring etcd data

Allows the operator to restore the data from a cluster-data snapshot. This comes with caveats and a very specific path to restore a cluster:

The cluster must be in a state of only having a single member. So it's best to deploy a new cluster using the etcd charm, without adding any additional units.

```
juju deploy etcd new-etcd
```

The above code snippet will deploy a single unit of etcd, as 'new-etcd'

```
juju run-action etcd/0 restore target=/mnt/etcd-backups
```

Once the restore action has completed, evaluate the cluster health. If the unit is healthy, you may resume scaling the application to meet your needs.

- **param** target: destination directory to save the existing data.
- **param** skip-backup: Don't backup any existing data.

Snapshot etcd data

Allows the operator to snapshot a running clusters data for use in cloning, backing up, or migrating Etcd clusters.

```
juju run-action etcd/0 snapshot target=/mnt/etcd-backups
```

- **param** target: destination directory to save the resulting snapshot archive.

Known Limitations

Loss of PKI warning

If you destroy the leader - identified with the `*` text next to the unit number in status: all TLS pki will be lost. No PKI migration occurs outside of the units requesting and registering the certificates. Important: Mismanaging this configuration will result in locking yourself out of the cluster, and can potentially break existing deployments in very strange ways relating to x509 validation of certificates, which affects both servers and clients.

Restoring from snapshot on a scaled cluster

Restoring from a snapshot on a scaled cluster will result in a broken cluster. Etcd performs clustering during unit turn-up, and state is stored in Etcd itself. During the snapshot restore phase, a new cluster ID is initialized, and peers are dropped from the snapshot state to enable snapshot restoration. Please follow the migration instructions above in the restore action description.

Upgrades

This page will outline how to manage and execute a Kubernetes upgrade.

Before you begin

This page assumes you have a working deployed cluster.

Assumptions

You should always back up all your data before attempting an upgrade. Don't forget to include the workload inside your cluster! Refer to the backup documentation.

Preparing for an Upgrade

See if upgrades are available. The Kubernetes charms are updated bi-monthly and mentioned in the Kubernetes release notes. Important operational considerations and change in behaviour will always be documented in the release notes.

You can use `juju status` to see if an upgrade is available. There will either be an upgrade to kubernetes or etcd, or both.

Upgrade etcd

Backing up etcd requires an export and snapshot, refer to the backup documentation to create a snapshot. After the snapshot upgrade the etcd service with:

```
juju upgrade-charm etcd
```

This will handle upgrades between minor versions of etcd. Major upgrades from etcd 2.x to 3.x are currently unsupported. Instead, data will be run in etcdv2 stores over the etcdv3 api.

Upgrade Kubernetes

The Kubernetes Charms use snap channels to drive payloads. The channels are defined by `X.Y/channel` where `X.Y` is the `major.minor` release of Kubernetes (e.g. 1.6) and `channel` is one of the four following channels:

Channel name	Description
stable	The latest stable released patch version of Kubernetes
candidate	Release candidate releases of Kubernetes
beta	Latest alpha or beta of Kubernetes for that minor release
edge	Nightly builds of that minor release of Kubernetes

If a release isn't available, the next highest channel is used. For example, 1.6/beta will load `/candidate` or `/stable` depending on availability of release. Development versions of Kubernetes are available in that minor releases edge channel. There is no guarantee that edge or master will work with the current charms.

Master Upgrades

First you need to upgrade the masters:

```
juju upgrade-charm kubernetes-master
```

NOTE: Always upgrade the masters before the workers.

Once the latest charm is deployed, the channel for Kubernetes can be selected by issuing the following:

```
juju config kubernetes-master channel=1.x/stable
```

Where `x` is the minor version of Kubernetes. For example, `1.6/stable`. See above for Channel definitions

Worker Upgrades

Two methods of upgrading workers are supported. Blue/Green Deployment and upgrade-in-place. Both methods are provided for operational flexibility and both are supported and tested. Blue/Green will require more hardware up front than inplace, but is a safer upgrade route.

Blue/Green Upgrade

Given the following deployment, where the workers are named kubernetes-alpha.

Deploy new worker(s):

```
juju deploy kubernetes-beta
```

Pause the old workers so your workload migrates:

```
juju run-action kubernetes-alpha/# pause
```

Verify old workloads have migrated with:

```
kubectl get pod -o wide
```

Tear down old workers with:

```
juju remove-application kubernetes-alpha
```

In place worker upgrade

```
juju upgrade-charm kubernetes-worker
juju config kubernetes-worker channel=1.x/stable
```

Where `x` is the minor version of Kubernetes. For example, `1.6/stable`. See above for Channel definitions. Once you've configured kubernetes-worker with the appropriate channel, run the upgrade action on each worker:

```
juju run-action kubernetes-worker/0 upgrade
juju run-action kubernetes-worker/1 upgrade
...
```

Verify upgrade

`kubectl version` should return the newer version.

It is recommended to rerun a cluster validation to ensure that the cluster upgrade has successfully completed.

Upgrade Flannel

Upgrading flannel can be done at any time, it is independent of Kubernetes upgrades. Be advised that networking is interrupted during the upgrade. You can initiate a flannel upgrade:

```
juju upgrade-charm flannel
```

Upgrade easyrsa

Upgrading easyrsa can be done at any time, it is independent of Kubernetes upgrades. Upgrading easyrsa should result in zero downtime as it is not a running service:

```
juju upgrade-charm easyrsa
```

Rolling back etcd

At this time rolling back etcd is unsupported.

Rolling back Kubernetes

At this time rolling back Kubernetes is unsupported.

Scaling

Any of the applications can be scaled out post-deployment. The charms update the status messages with progress, so it is recommended to run.

```
watch -c juju status --color
```

This page shows how to horizontally scale master and worker nodes on a cluster.

Before you begin

This page assumes you have a working Juju deployed cluster.

Kubernetes masters

The provided Kubernetes master nodes act as a control plane for the cluster. The deployment has been designed so that these nodes can be scaled independently of worker nodes to allow for more operational flexibility. To scale a master node up, simply execute:

```
juju add-unit kubernetes-master
```

This will add another master node to the control plane. See the building high-availability clusters section of the documentation for more information.

Kubernetes workers

The kubernetes-worker nodes are the load-bearing units of a Kubernetes cluster.

By default pods are automatically spread throughout the kubernetes-worker units that you have deployed.

To add more kubernetes-worker units to the cluster:

```
juju add-unit kubernetes-worker
```

or specify machine constraints to create larger nodes:

```
juju set-constraints kubernetes-worker "cpu-cores=8 mem=32G"
juju add-unit kubernetes-worker
```

Refer to the machine constraints documentation for other machine constraints that might be useful for the kubernetes-worker units.

etcd

Etcd is used as a key-value store for the Kubernetes cluster. The bundle defaults to one instance in this cluster.

For quorum reasons it is recommended to keep an odd number of etcd nodes. 3, 5, 7, and 9 nodes are the recommended amount of nodes, depending on your cluster size. The CoreOS etcd documentation has a chart for the optimal cluster size to determine fault tolerance.

To add an etcd unit:

```
juju add-unit etcd
```

Shrinking of an etcd cluster after growth is not recommended.

Juju Controller

A single node is responsible for coordinating with all the Juju agents on each machine that manage Kubernetes, it is called the controller node. For production deployments it is recommended to enable HA of the controller node:

```
juju enable-ha
```

Enabling HA results in 3 controller nodes, this should be sufficient for most use cases. 5 and 7 controller nodes are also supported for extra large deployments.

Refer to the Juju HA controller documentation for more information.

Setting up Kubernetes with Juju

Out of the box it comes with the following components on 9 machines:

- Kubernetes (automated deployment, operations, and scaling)
    - Three node Kubernetes cluster with one master and two worker nodes.
    - TLS used for communication between units for security.
    - Flannel Software Defined Network (SDN) plugin
    - A load balancer for HA kubernetes-master (Experimental)
    - Optional Ingress Controller (on worker)
    - Optional Dashboard addon (on master) including Heapster for cluster monitoring
- EasyRSA
    - Performs the role of a certificate authority serving self signed certificates to the requesting units of the cluster.
- ETCD (distributed key value store)
    - Three unit cluster for reliability.

The Juju Kubernetes work is curated by a dedicated team of community members, let us know how we are doing. If you find any problems please open an issue on our tracker so we can find them.

Support Level

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Amazon Web Services (AWS)	Juju	Ubuntu	flannel, calico*	docs		Commercial, Community (@mbruzek, @chuckbutler)
OpenStack	Juju	Ubuntu	flannel, calico	docs		Commercial, Community (@mbruzek, @chuckbutler)

IaaS Provider	Config. Mgmt	OS	Networking	Docs	Conforms	Support Level
Microsoft Azure	Juju	Ubuntu	flannel	docs		Commercial, Community (@mbruzek, @chuckbutler)
Google Compute Engine (GCE)	Juju	Ubuntu	flannel, calico	docs		Commercial, Community (@mbruzek, @chuckbutler)
Joyent	Juju	Ubuntu	flannel	docs		Commercial, Community (@mbruzek, @chuckbutler)
Rackspace	Juju	Ubuntu	flannel	docs		Commercial, Community (@mbruzek, @chuckbutler)
VMWare vSphere	Juju	Ubuntu	flannel, calico	docs		Commercial, Community (@mbruzek, @chuckbutler)
Bare Metal (MAAS)	Juju	Ubuntu	flannel, calico	docs		Commercial, Community (@mbruzek, @chuckbutler)

For support level information on all solutions, see the Table of solutions chart.

Ubuntu 16.04 introduced the Canonical Distribution of Kubernetes, a pure upstream distribution of Kubernetes designed for production usage. This page shows you how to deploy a cluster.

Before you begin

Prerequisites

- A working Juju client; this does not have to be a Linux machine, it can also be Windows or OSX.
- A supported cloud.

  - Bare Metal deployments are supported via MAAS. Refer to the MAAS documentation for configuration instructions.
  - OpenStack deployments are currently only tested on Icehouse and newer.
- Network access to the following domains
  - *.jujucharms.com
  - gcr.io
  - github.com
  - Access to an Ubuntu mirror (public or private)

Configure Juju to use your cloud provider

Deployment of the cluster is supported on a wide variety of public clouds, private OpenStack clouds, or raw bare metal clusters. Bare metal deployments are supported via MAAS.

After deciding which cloud to deploy to, follow the cloud setup page to configure deploying to that cloud.

Load your cloud credentials for each cloud provider you would like to use.

In this example

```
juju add-credential aws
credential name: my_credentials
select auth-type [userpass, oauth, etc]: userpass
enter username: jorge
enter password: *******
```

You can also just auto load credentials for popular clouds with the `juju autoload-credentials` command, which will auto import your credentials from the default files and environment variables for each cloud.

Next we need to bootstrap a controller to manage the cluster. You need to define the cloud you want to bootstrap on, the region, and then any name for your controller node:

```
juju update-clouds # This command ensures all the latest regions are up to date on
juju bootstrap aws/us-east-2
```

or, another example, this time on Azure:

```
juju bootstrap azure/centralus
```

You will need a controller node for each cloud or region you are deploying to. See the controller documentation for more information.

Note that each controller can host multiple Kubernetes clusters in a given cloud or region.

Launch a Kubernetes cluster

The following command will deploy the initial 9-node starter cluster. The speed of execution is very dependent of the performance of the cloud you're deploying to:

```
juju deploy canonical-kubernetes
```

After this command executes the cloud will then launch instances and begin the deployment process.

Monitor deployment

The `juju status` command provides information about each unit in the cluster. Use the `watch -c juju status --color` command to get a real-time view of the cluster as it deploys. When all the states are green and "Idle", the cluster is ready to be used:

```
juju status
```

Output:

```
Model    Controller     Cloud/Region   Version
default  aws-us-east-2  aws/us-east-2  2.0.1

App                     Version  Status       Scale  Charm                   Store
easyrsa                 3.0.1    active           1  easyrsa                 jujucha
etcd                    3.1.2    active           3  etcd                    jujucha
flannel                 0.6.1    maintenance      4  flannel                 jujucha
kubeapi-load-balancer   1.10.0   active           1  kubeapi-load-balancer   jujucha
kubernetes-master       1.6.1    active           1  kubernetes-master       jujucha
kubernetes-worker       1.6.1    active           3  kubernetes-worker       jujucha
topbeat                          active           3  topbeat                 jujucha

Unit                      Workload     Agent  Machine  Public address  Ports
easyrsa/0*                active       idle   0        52.15.95.92
etcd/0                    active       idle   3        52.15.79.127    2379/tcp
etcd/1*                   active       idle   4        52.15.111.66    2379/tcp
etcd/2                    active       idle   5        52.15.144.25    2379/tcp
kubeapi-load-balancer/0*  active       idle   7        52.15.84.179    443/tcp
kubernetes-master/0*      active       idle   8        52.15.106.225   6443/tcp
  flannel/3               active       idle            52.15.106.225
kubernetes-worker/0*      active       idle   9        52.15.153.246
  flannel/2               active       idle            52.15.153.246
kubernetes-worker/1       active       idle   10       52.15.52.103
  flannel/0*              active       idle            52.15.52.103
kubernetes-worker/2       active       idle   11       52.15.104.181
  flannel/1               active       idle            52.15.104.181

Machine  State    DNS            Inst id              Series  AZ
0        started  52.15.95.92    i-06e66414008eca61c  xenial  us-east-2c
3        started  52.15.79.127   i-0038186d2c5103739  xenial  us-east-2b
4        started  52.15.111.66   i-0ac66c86a8ec93b18  xenial  us-east-2a
5        started  52.15.144.25   i-078cfe79313d598c9  xenial  us-east-2c
7        started  52.15.84.179   i-00fd70321a51b658b  xenial  us-east-2c
8        started  52.15.106.225  i-0109a5fc942c53ed7  xenial  us-east-2b
9        started  52.15.153.246  i-0ab63e34959cace8d  xenial  us-east-2b
10       started  52.15.52.103   i-0108a8cc0978954b5  xenial  us-east-2a
11       started  52.15.104.181  i-0f5562571c649f0f2  xenial  us-east-2c
```

Interacting with the cluster

After the cluster is deployed you may assume control over the cluster from any kubernetes-master, or kubernetes-worker node.

First you need to download the credentials and client application to your local workstation:

Create the kubectl config directory.

```
mkdir -p ~/.kube
```

Copy the kubeconfig file to the default location.

```
juju scp kubernetes-master/0:config ~/.kube/config
```

Fetch a binary for the architecture you have deployed. If your client is a different architecture you will need to get the appropriate `kubectl` binary through other means. In this example we copy kubectl to `~/bin` for convenience, by default this should be in your $PATH.

```
mkdir -p ~/bin
juju scp kubernetes-master/0:kubectl ~/bin/kubectl
```

Query the cluster:

```
kubectl cluster-info
```

Output:

```
Kubernetes master is running at https://52.15.104.227:443
Heapster is running at https://52.15.104.227:443/api/v1/namespaces/kube-system/ser
KubeDNS is running at https://52.15.104.227:443/api/v1/namespaces/kube-system/serv
Grafana is running at https://52.15.104.227:443/api/v1/namespaces/kube-system/serv
InfluxDB is running at https://52.15.104.227:443/api/v1/namespaces/kube-system/ser
```

Congratulations, you've now set up a Kubernetes cluster!

Scale up cluster

Want larger Kubernetes nodes? It is easy to request different sizes of cloud resources from Juju by using **constraints**. You can increase the amount of CPU or memory (RAM) in any of the systems requested by Juju. This allows you to fine tune the Kubernetes cluster to fit your workload. Use flags on the bootstrap command or as a separate `juju constraints` command. Look to the Juju documentation for machine details.

Scale out cluster

Need more workers? We just add more units:

```
juju add-unit kubernetes-worker
```

Or multiple units at one time:

```
juju add-unit -n3 kubernetes-worker
```

You can also ask for specific instance types or other machine-specific constraints. See the constraints documentation for more information. Here are some examples, note that generic constraints such as `cores` and `mem` are more portable between clouds. In this case we'll ask for a specific instance type from AWS:

```
juju set-constraints kubernetes-worker instance-type=c4.large
juju add-unit kubernetes-worker
```

You can also scale the etcd charm for more fault tolerant key/value storage:

```
juju add-unit -n3 etcd
```

It is strongly recommended to run an odd number of units for quorum.

Tear down cluster

If you want stop the servers you can destroy the Juju model or the controller. Use the `juju switch` command to get the current controller name:

```
juju switch
juju destroy-controller $controllername --destroy-all-models
```

This will shutdown and terminate all running instances on that cloud.

More Info

The Ubuntu Kubernetes deployment uses open-source operations, or operations as code, known as charms. These charms are assembled from layers which keeps the code smaller and more focused on the operations of just Kubernetes and its components.

The Kubernetes layer and bundles can be found in the `kubernetes` project on github.com:

- Bundle location
- Kubernetes charm layer location
- Canonical Kubernetes home

Feature requests, bug reports, pull requests or any feedback would be much appreciated.

Monitoring

This page shows how to connect various logging solutions to a Juju deployed cluster.

Before you begin

This page assumes you have a working Juju deployed cluster.

Connecting Datadog

Datadog is a SaaS offering which includes support for a range of integrations, including Kubernetes and ETCD. While the solution is SAAS/Commercial, they include a Free tier which is supported with the following method. To deploy a full Kubernetes stack with Datadog out of the box, do:

```
juju deploy canonical-kubernetes-datadog
```

Installation of Datadog

To start, deploy the latest version Datadog from the Charm Store:

```
juju deploy datadog
```

Configure Datadog with your api-key, found in the Datadog dashboard. Replace `XXXX` with your API key.

```
juju configure datadog api-key=XXXX
```

Finally, attach `datadog` to all applications you wish to monitor. For example, kubernetes-master, kubernetes-worker, and etcd:

```
juju add-relation datadog kubernetes-worker
juju add-relation datadog kubernetes-master
juju add-relation datadog etcd
```

Connecting Elastic stack

The Elastic stack, formally "ELK" stack, refers to Elastic Search and the suite of tools to facilitate log aggregation, monitoring, and dashboarding. To deploy a full Kubernetes stack with elastic out of the box, do:

```
juju deploy canonical-kubernetes-elastic
```

New install of ElasticSearch

To start, deploy the latest version of ElasticSearch, Kibana, Filebeat, and Topbeat from the Charm Store:

This can be done in one command as:

```
juju deploy beats-core
```

However, if you wish to customize the deployment, or proceed manually, the following commands can be issued:

```
juju deploy elasticsearch
juju deploy kibana
juju deploy filebeat
juju deploy topbeat

juju add-relation elasticsearch kibana
juju add-relation elasticsearch topbeat
juju add-relation elasticsearch filebeat
```

Finally, connect filebeat and topbeat to all applications you wish to monitor. For example, kubernetes-master and kubernetes-worker:

```
juju add-relation kubernetes-master topbeat
juju add-relation kubernetes-master filebeat
juju add-relation kubernetes-worker topbeat
juju add-relation kubernetes-worker filebeat
```

Existing ElasticSearch cluster

In the event an ElasticSearch cluster already exists, the following can be used to connect and leverage it instead of creating a new, separate, cluster. First deploy the two beats, filebeat and topbeat

```
juju deploy filebeat
juju deploy topbeat
```

Configure both filebeat and topbeat to connect to your ElasticSearch cluster, replacing `255.255.255.255` with the IP address in your setup.

```
juju configure filebeat elasticsearch=255.255.255.255
juju configure topbeat elasticsearch=255.255.255.255
```

Follow the above instructions on connect topbeat and filebeat to the applications you wish to monitor.

Connecting Nagios

Nagios utilizes the Nagions Remote Execution Protocol (NRPE) as an agent on each node to derive machine level details of the health and applications.

New install of Nagios

To start, deploy the latest version of the Nagios and NRPE charms from the store:

```
juju deploy nagios
juju deploy nrpe
```

Connect Nagios to NRPE

```
juju add-relation nagios nrpe
```

Finally, add NRPE to all applications deployed that you wish to monitor, for example `kubernetes-master`, `kubernetes-worker`, `etcd`, `easyrsa`, and `kubeapi-load-balancer`.

```
juju add-relation nrpe kubernetes-master
juju add-relation nrpe kubernetes-worker
juju add-relation nrpe etcd
juju add-relation nrpe easyrsa
juju add-relation nrpe kubeapi-load-balancer
```

Existing install of Nagios

If you already have an existing Nagios installation, the `nrpe-external-master` charm can be used instead. This will allow you to supply configuration options that map your existing external Nagios installation to NRPE. Replace `255.255.255.255` with the IP address of the nagios instance.

```
juju deploy nrpe-external-master
juju configure nrpe-external-master nagios_master=255.255.255.255
```

Once configured, connect nrpe-external-master as outlined above.

Networking

Kubernetes supports the Container Network Interface (CNI). This is a network plugin architecture that allows you to use whatever Kubernetes-friendly SDN you want. Currently this means support for Flannel.

This page shows how to the various network portions of a cluster work, and how to configure them.

Before you begin

This page assumes you have a working Juju deployed cluster.

Flannel

Usage

The flannel charm is a subordinate. This charm will require a principal charm that implements the `kubernetes-cni` interface in order to properly deploy.

```
juju deploy flannel
juju deploy etcd
juju deploy kubernetes-master
juju add-relation flannel kubernetes-master
juju add-relation flannel etcd
```

Configuration

iface The interface to configure the flannel SDN binding. If this value is empty string or undefined the code will attempt to find the default network adapter similar to the following command:

```
$ route | grep default | head -n 1 | awk {'print $8'}
```

cidr The network range to configure the flannel SDN to declare when establishing networking setup with etcd. Ensure this network range is not active on layers 2/3 you're deploying to, as it will cause collisions and odd behavior if care is not taken when selecting a good CIDR range to assign to flannel. It's also good practice to ensure you allot yourself a large enough IP range to support how large your cluster will potentially scale. Class A IP ranges with /24 are a good option.

Security Considerations

By default all connections between every provided node is secured via TLS by easyrsa, including the etcd cluster.

Implementation

The TLS and easyrsa implementations use the following layers.

layer-tls-client layer-easyrsa

This page explains the security considerations of a deployed cluster and production recommendations.

Before you begin

This page assumes you have a working Juju deployed cluster.

Limiting ssh access

By default the administrator can ssh to any deployed node in a cluster. You can mass disable ssh access to the cluster nodes by issuing the following command.

```
juju model-config proxy-ssh=true
```

Note: The Juju controller node will still have open ssh access in your cloud, and will be used as a jump host in this case.

Refer to the model management page in the Juju documentation for instructions on how to manage ssh keys.

Storage

This page explains how to install and configure persistent storage on a cluster.

- Before you begin
- Ceph Persistent Volumes

Before you begin

This page assumes you have a working Juju deployed cluster.

Ceph Persistent Volumes

The Canonical Distribution of Kubernetes allows you to connect with durable storage devices such as Ceph. When paired with the Juju Storage feature you can add durable storage easily and across clouds.

Deploy a minimum of three ceph-mon and three ceph-osd units.

```
juju deploy cs:ceph-mon -n 3
juju deploy cs:ceph-osd -n 3
```

Relate the units together: `juju add-relation ceph-mon ceph-osd`

List the storage pools available to Juju for your cloud:

```
juju storage-pools
```

Output:

```
Name Provider Attrs ebs ebs ebs-ssd ebs volume-type=ssd loop loop rootfs rootfs
tmpfs tmpfs
```

> **Note**: This listing is for the Amazon Web Services public cloud. > Different clouds may have different pool names.

Add a storage pool to the ceph-osd charm by NAME,SIZE,COUNT:

```
juju add-storage ceph-osd/0 osd-devices=ebs,10G,1
juju add-storage ceph-osd/1 osd-devices=ebs,10G,1
juju add-storage ceph-osd/2 osd-devices=ebs,10G,1
```

Next relate the storage cluster with the Kubernetes cluster:

```
juju add-relation kubernetes-master ceph-mon
```

We are now ready to enlist Persistent Volumes in Kubernetes which our workloads can consume via Persistent Volume (PV) claims.

```
juju run-action kubernetes-master/0 create-rbd-pv name=test size=50
```

This example created a “test” Radios Block Device (rbd) in the size of 50 MB.

Use watch on your Kubernetes cluster like the following, you should see the PV become enlisted and be marked as available:

```
watch kubectl get pv
```

Output:

```
NAME  CAPACITY   ACCESSMODES   STATUS    CLAIM              REASON    AGE

test   50M          RWO       Available                               10s
```

To consume these Persistent Volumes, your pods will need an associated Persistent Volume Claim with them, and is outside the scope of this README. See the Persistent Volumes documentation for more information.

Troubleshooting

This document with highlighting how to troubleshoot the deployment of a Kubernetes cluster, it will not cover debugging of workloads inside Kubernetes.

Before you begin

This page assumes you have a working Juju deployed cluster.

Understanding Cluster Status

Using `juju status` can give you some insight as to what's happening in a cluster:

```
Model  Controller  Cloud/Region   Version
kubes  work-multi  aws/us-east-2  2.0.2.1

App                Version  Status  Scale  Charm              Store       Rev  OS
easyrsa            3.0.1    active      1  easyrsa            jujucharms    3  ubu
etcd               2.2.5    active      1  etcd               jujucharms   17  ubu
flannel            0.6.1    active      2  flannel            jujucharms    6  ubu
kubernetes-master  1.4.5    active      1  kubernetes-master  jujucharms    8  ubu
kubernetes-worker  1.4.5    active      1  kubernetes-worker  jujucharms   11  ubu

Unit                 Workload  Agent  Machine  Public address  Ports           Me
easyrsa/0*           active    idle   0/lxd/0  10.0.0.55                       Ce
etcd/0*              active    idle   0        52.15.47.228    2379/tcp        He
kubernetes-master/0*  active   idle   0        52.15.47.228    6443/tcp        Ku
  flannel/1          active    idle            52.15.47.228                    Fl
kubernetes-worker/0*  active   idle   1        52.15.177.233   80/tcp,443/tcp  Ku
  flannel/0*         active    idle            52.15.177.233                   Fl

Machine  State    DNS            Inst id              Series  AZ
0        started  52.15.47.228   i-0bb211a18be691473  xenial  us-east-2a
0/lxd/0  started  10.0.0.55      juju-153b74-0-lxd-0  xenial
1        started  52.15.177.233  i-0502d7de733be31bb  xenial  us-east-2b
```

In this example we can glean some information. The `Workload` column will show the status of a given service. The `Message` section will show you the health of a given service in the cluster. During deployment and maintenance these workload statuses will update to reflect what a given node is doing. For example the workload my say `maintenance` while message will describe this maintenance as `Installing docker`.

During normal operation the Workload should read `active`, the Agent column (which reflects what the Juju agent is doing) should read `idle`, and the messages will either say `Ready` or another descriptive term. `juju status --color` will also return all green results when a cluster's deployment is healthy.

Status can become unwieldy for large clusters, it is then recommended to check status on individual services, for example to check the status on the workers only:

```
juju status kubernetes-workers
```

or just on the etcd cluster:

```
juju status etcd
```

Errors will have an obvious message, and will return a red result when used with `juju status --color`. Nodes that come up in this manner should be investigated.

SSHing to units.

You can ssh to individual units easily with the following convention, `juju ssh /<unit#>:

```
juju ssh kubernetes-worker/3
```

Will automatically ssh you to the 3rd worker unit.

```
juju ssh easyrsa/0
```

This will automatically ssh you to the easyrsa unit.

Collecting Debug information

Sometimes it is useful to collect all the information from a node to share with a developer so problems can be identifying. This section will deal on how to use the debug action to collect this information. The debug action is only supported on `kubernetes-worker` nodes.

```
juju run-action kubernetes-worker/0 debug
```

Which returns:

```
Action queued with id: 4b26e339-7366-4dc7-80ed-255ac0377020`
```

This produces a .tar.gz file which you can retrieve:

```
juju show-action-output 4b26e339-7366-4dc7-80ed-255ac0377020
```

This will give you the path for the debug results:

```
results:
  command: juju scp debug-test/0:/home/ubuntu/debug-20161110151539.tar.gz .
  path: /home/ubuntu/debug-20161110151539.tar.gz
status: completed
timing:
  completed: 2016-11-10 15:15:41 +0000 UTC
  enqueued: 2016-11-10 15:15:38 +0000 UTC
  started: 2016-11-10 15:15:40 +0000 UTC
```

You can now copy the results to your local machine:

```
juju scp kubernetes-worker/0:/home/ubuntu/debug-20161110151539.tar.gz .
```

The archive includes basic information such as systemctl status, Juju logs, charm unit data, etc. Additional application-specific information may be included as well.

Common Problems

Load Balancer interfering with Helm

This section assumes you have a working deployment of Kubernetes via Juju using a Load Balancer for the API, and that you are using Helm to deploy charts.

To deploy Helm you will have run:

```
helm init
$HELM_HOME has been configured at /home/ubuntu/.helm
Tiller (the helm server side component) has been installed into your Kubernetes Cl
Happy Helming!
```

Then when using helm you may see one of the following errors:

- Helm doesn't get the version from the Tiller server

```
helm version
Client: &version.Version{SemVer:"v2.1.3", GitCommit:"5cbc48fb305ca4bf68c26eb8d2a7e
Error: cannot connect to Tiller
```

- Helm cannot install your chart

```
helm install <chart> --debug
Error: forwarding ports: error upgrading connection: Upgrade request required
```

This is caused by the API load balancer not forwarding ports in the context of the helm client-server relationship. To deploy using helm, you will need to follow these steps:

1. Expose the Kubernetes Master service

```
juju expose kubernetes-master
```

1. Identify the public IP address of one of your masters

```
juju status kubernetes-master
Model       Controller  Cloud/Region   Version
production  k8s-admin   aws/us-east-1  2.0.0

App                Version  Status  Scale  Charm              Store       Rev  OS
flannel            0.6.1    active      1  flannel            jujucharms    7  ubu
kubernetes-master  1.5.1    active      1  kubernetes-master  jujucharms   10  ubu

Unit                  Workload  Agent  Machine  Public address  Ports     Message
kubernetes-master/0*  active    idle   5        54.210.100.102    6443/tcp  Kubern
  flannel/0           active    idle            54.210.100.102              Flanne

Machine  State    DNS             Inst id              Series   AZ
5        started  54.210.100.102  i-002b7150639eb183b  xenial  us-east-1a

Relation      Provides               Consumes           Type
certificates  easyrsa                kubernetes-master  regular
etcd          etcd                   flannel            regular
etcd          etcd                   kubernetes-master  regular
cni           flannel                kubernetes-master  regular
loadbalancer  kubeapi-load-balancer  kubernetes-master  regular
cni           kubernetes-master      flannel            subordinate
cluster-dns   kubernetes-master      kubernetes-worker  regular
cni           kubernetes-worker      flannel            subordinate
```

In this context the public IP address is 54.210.100.102.

If you want to access this data programmatically you can use the JSON output:

```
juju show-status kubernetes-master --format json | jq --raw-output '.applications.
54.210.100.102
```

1. Update the kubeconfig file

Identify the kubeconfig file or section used for this cluster, and edit the server configuration.

By default, it will look like `https://54.213.123.123:443`. Replace it with the Kubernetes Master endpoint `https://54.210.100.102:6443` and save.

Note that the default port used by CDK for the Kubernetes Master API is 6443 while the port exposed by the load balancer is 443.

1. Start helming again!

```
helm install <chart> --debug
Created tunnel using local port: '36749'
SERVER: "localhost:36749"
CHART PATH: /home/ubuntu/.helm/<chart>
NAME:   <chart>
...
...
```

etcd

Kubernetes

By default there is no log aggregation of the Kubernetes nodes, each node logs locally. It is recommended to deploy the Elastic Stack for log aggregation if you desire centralized logging.

Decommissioning

Warning: By the time you've reached this step you should have backed up your workloads and pertinent data, this section is for the complete destruction of a cluster.

This page shows you how to properly decommission a cluster.

Before you begin

This page assumes you have a working Juju deployed cluster.

It is recommended to deploy individual Kubernetes clusters in their own models, so that there is a clean separation between environments. To remove a cluster first find out which model it's in with `juju list-models`. The controller reserves an `admin` model for itself. If you have chosen to not name your model it might show up as `default`.

```
$ juju list-models
Controller: aws-us-east-2

Model       Cloud/Region   Status     Machines  Cores  Access  Last connection
controller  aws/us-east-2  available         1      2  admin   just now
my-kubernetes-cluster*    aws/us-east-2  available        12     22  admin   2 min
```

You can then destroy the model, which will in turn destroy the cluster inside of it:

```
juju destroy-model my-kubernetes-cluster
```

```
$ juju destroy-model my-kubernetes-cluster
WARNING! This command will destroy the "my-kubernetes-cluster" model.
This includes all machines, applications, data and other resources.

Continue [y/N]? y
Destroying model
Waiting on model to be removed, 12 machine(s), 10 application(s)...
Waiting on model to be removed, 12 machine(s), 9 application(s)...
Waiting on model to be removed, 12 machine(s), 8 application(s)...
Waiting on model to be removed, 12 machine(s), 7 application(s)...
Waiting on model to be removed, 12 machine(s)...
Waiting on model to be removed...
$
```

This will destroy and decommission all nodes. You can confirm all nodes are destroyed by running `juju status`.

If you're using a public cloud this will terminate the instances. If you're on bare metal using MAAS this will release the nodes, optionally wipe the disk, power off the machines, and return them to available pool of machines to deploy from.

Cleaning up the Controller

If you're not using the controller for anything else, you will also need to remove the controller instance:

```
$ juju list-controllers
Use --refresh flag with this command to see the latest information.

Controller      Model  User   Access     Cloud/Region   Models  Machines    HA  Ve
aws-us-east-2*  -      admin  superuser  aws/us-east-2       2         1  none  2.

$ juju destroy-controller aws-us-east-2
WARNING! This command will destroy the "aws-us-east-2" controller.
This includes all machines, applications, data and other resources.

Continue? (y/N):y
Destroying controller
Waiting for hosted model resources to be reclaimed
All hosted models reclaimed, cleaning up controller machines
$
```

Operational Considerations

This page gives recommendations and hints for people managing long lived clusters

Before you begin

This page assumes you understand the basics of Juju and Kubernetes.

Managing Juju

Sizing your controller node

The Juju Controller:

- requires about 2 to 2.5GB RAM to operate.
- uses a MongoDB database as a storage backend for the configuration and state of the cluster. This database can grow significantly, and can also be the biggest consumer of CPU cycles on the instance
- aggregates and stores the log data of all services and units. Therefore, significant storage is needed for long lived models. If your intention is to keep the cluster running, make sure to provision at least 64GB for the logs.

To bootstrap a controller with constraints run the following command:

```
juju bootstrap --contraints "mem=8GB cpu-cores=4 root-disk=128G"
```

Juju will select the cheapest instance type matching your constraints on your target cloud. You can also use the `instance-type` constraint in conjunction with `root-disk` for strict control. For more information about the constraints available, refer to the official documentation

Additional information about logging can be found in the logging section

SSHing into the Controller Node

By default, Juju will create a pair of SSH keys that it will use to automate the connection to units. They are stored on the client node in `~/.local/share/juju/ssh/`

After deployment, Juju Controller is a “silent unit” that acts as a proxy between the client and the deployed applications. Nevertheless it can be useful to SSH into it.

First you need to understand your environment, especially if you run several Juju models and controllers. Run

```
juju list-models --all
$ juju models --all
Controller: k8s

Model             Cloud/Region   Status     Machines  Cores  Access  Last connecti
admin/controller  lxd/localhost  available         1      -  admin   just now
admin/default     lxd/localhost  available         0      -  admin   2017-01-23
admin/whale*      lxd/localhost  available         6      -  admin   3 minutes ago
```

The first line `Controller: k8s` refers to how you bootstrapped.

Then you will see 2, 3 or more models listed below.

- admin/controller is the default model that hosts all controller units of juju
- admin/default is created by default as the primary model to host the user application, such as the Kubernetes cluster
- admin/whale is an additional model created if you use conjure-up as an overlay on top of Juju.

Now to ssh into a controller node, you first ask Juju to switch context, then ssh as you would with a normal unit:

```
juju switch controller
```

At this stage, you can query the controller model as well:

```
juju status
Model       Controller  Cloud/Region   Version
controller  k8s           lxd/localhost  2.0.2

App  Version  Status  Scale  Charm  Store  Rev  OS  Notes

Unit  Workload  Agent  Machine  Public address  Ports  Message

Machine  State    DNS           Inst id        Series  AZ
0        started  10.191.22.15  juju-2a5ed8-0  xenial
```

Note that if you had bootstrapped in HA mode, you would see several machines listed.

Now ssh-ing into the controller follows the same semantic as classic Juju commands:

```
$ juju ssh 0
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.8.0-34-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


Last login: Tue Jan 24 16:38:13 2017 from 10.191.22.1
ubuntu@juju-2a5ed8-0:~$
```

When you are done and want to come back to your initial model, exit the controller and

Then if you need to switch back to your cluster and ssh into the units, run

```
juju switch default
```

Managing your Kubernetes cluster

Running privileged containers

By default, juju-deployed clusters do not support running privileged containers. If you need them, you have to enable the `allow-privileged` config on both kubernetes-master and kubernetes-worker:

```
juju config kubernetes-master allow-privileged=true
juju config kubernetes-worker allow-privileged=true
```

Private registry

With the registry action, you can easily create a private docker registry that uses TLS authentication. However, note that a registry deployed with that action is not HA; it uses storage tied to the kubernetes node where the pod is running. Consequently, if the registry pod is migrated from one node to another, you will need to re-publish the images.

Example usage

Create the relevant authentication files. Let’s say you want user `userA` to authenticate with the password `passwordA`. Then you’ll do:

```
echo "userA:passwordA" > htpasswd-plain
htpasswd -c -b -B htpasswd userA passwordA
```

(the `htpasswd` program comes with the `apache2-utils` package)

Assuming that your registry will be reachable at `myregistry.company.com`, you already have your TLS key in the `registry.key` file, and your TLS certificate (with `myregistry.company.com` as Common Name) in the `registry.crt` file, you would then run:

```
juju run-action kubernetes-worker/0 registry domain=myregistry.company.com htpassw
```

If you then decide that you want do delete the registry, just run:

```
juju run-action kubernetes-worker/0 registry delete=true ingress=true
```

Glossary and Terminology

This page explains some of the terminology used in deploying Kubernetes with Juju.

- Before you begin

Before you begin

This page assumes you have a working Juju deployed cluster.

controller - The management node of a cloud environment. Typically you have one controller per cloud region, or more in HA environments. The controller is responsible for managing all subsequent models in a given environment. It contains the Juju API server and its underlying database.

model - A collection of charms and their relationships that define a deployment. This includes machines and units. A controller can host multiple models. It is recommended to separate Kubernetes clusters into individual models for management and isolation reasons.

charm - The definition of a service, including its metadata, dependencies with other services, required packages, and application management logic. It contains all the operational knowledge of deploying a Kubernetes cluster. Included charm examples are `kubernetes-core` , `easy-rsa` , `kibana` , and `etcd` .

unit - A given instance of a service. These may or may not use up a whole machine, and may be colocated on the same machine. So for example you might have a `kubernetes-worker` , and `filebeat` , and `topbeat` units running on a single machine, but they are three distinct units of different services.

machine - A physical node, these can either be bare metal nodes, or virtual machines provided by a cloud.

Local Kubernetes development with LXD

The purpose of using LXD on a local machine is to emulate the same deployment that a user would use in a cloud or bare metal. Each node is treated as a machine, with the same characteristics as production. Each node is a separate container, which runs Docker containers and `kubectl` inside (see Cluster Intro for more info).

Running Kubernetes locally has obvious development advantages, such as lower cost and faster iteration than constantly deploying and tearing down clusters on a public cloud. Ideally, a Kubernetes developer can spawn all necessary nodes inside local containers and test new configurations as they are committed. This page will show you how to deploy a cluster to LXD containers on a local machine.

Before you begin

Install conjure-up, a tool for deploying big software.

```
sudo snap install conjure-up --classic
```

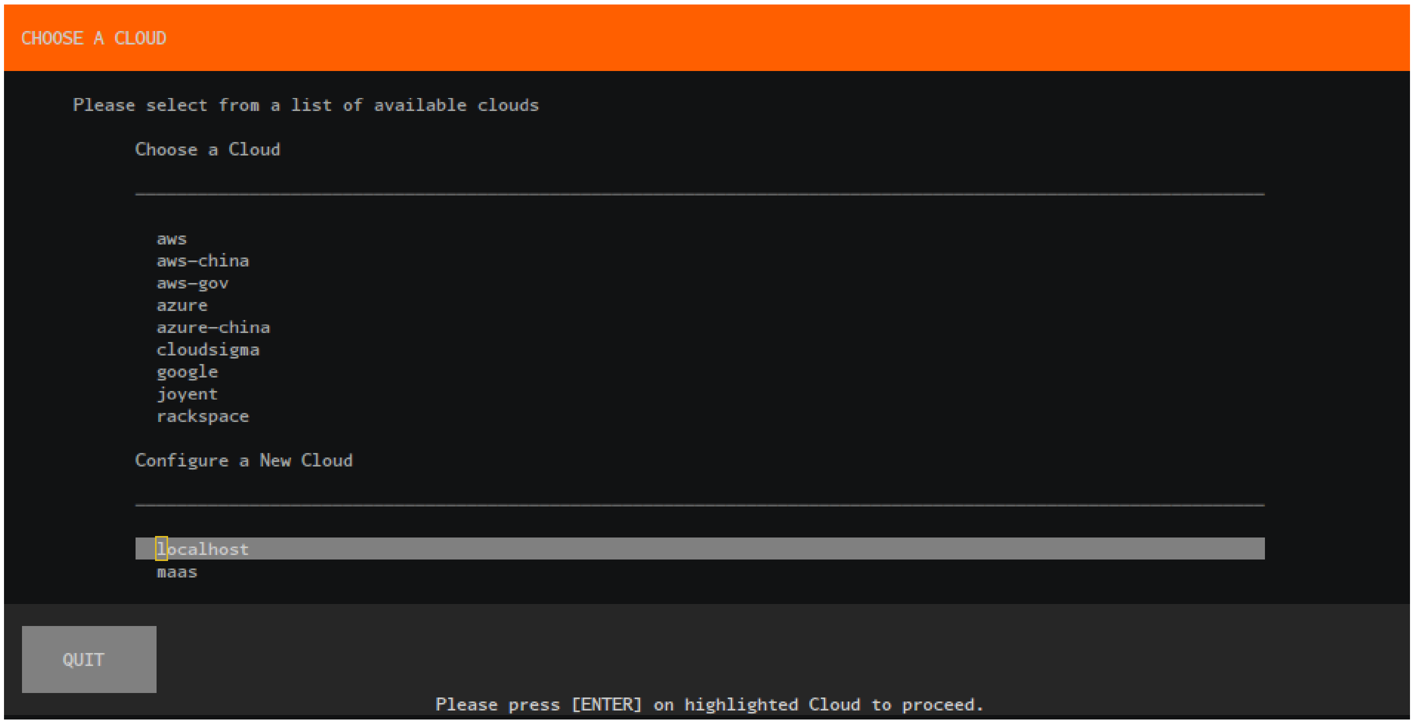
Note: If conjure-up asks you to “Setup an ipv6 subnet” with LXD, answer NO. ipv6 with Juju/LXD is currently unsupported.

Deploying Kubernetes

Start the deployment with:

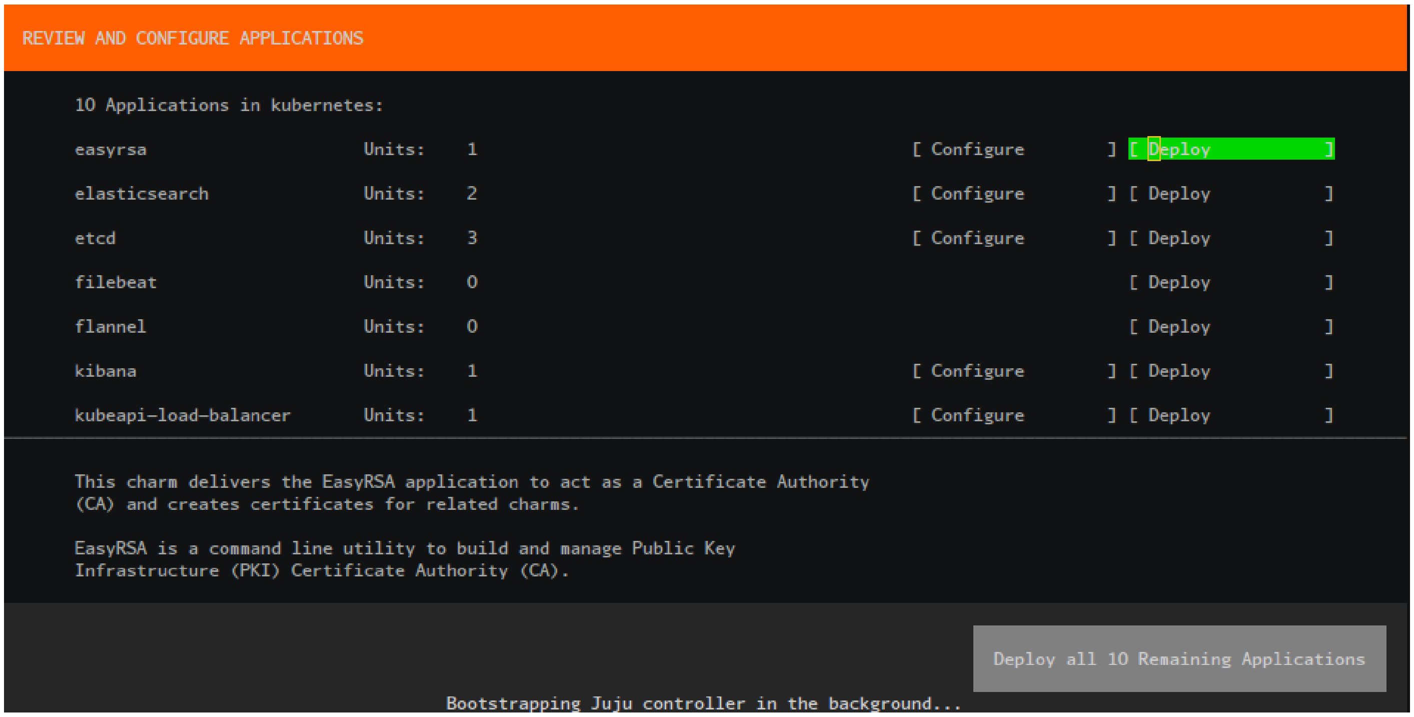
```
conjure-up kubernetes
```

For this walkthrough we are going to create a new controller - select the `localhost` Cloud type:



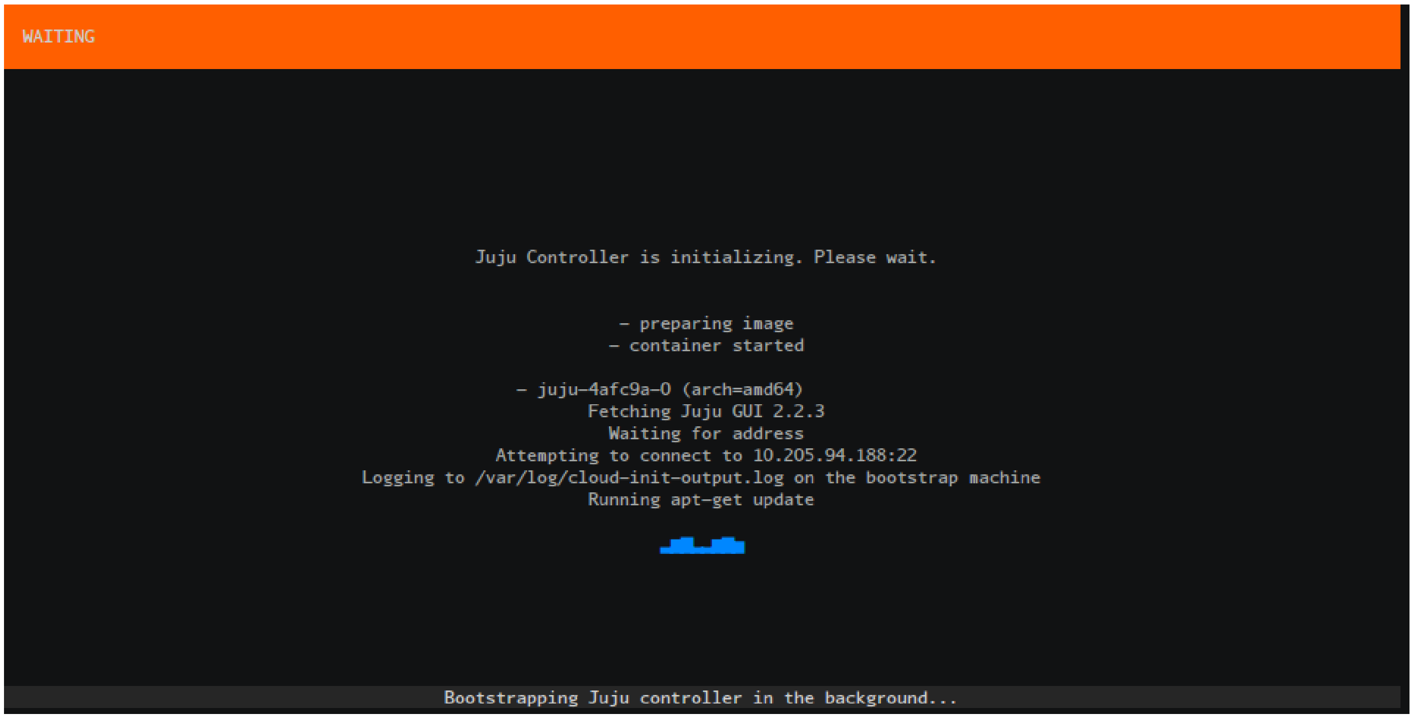
```
CHOOSE A CLOUD

     Please select from a list of available clouds

          Choose a Cloud

          ____________________________________________________________

            aws
            aws-china
            aws-gov
            azure
            azure-china
            cloudsigma
            google
            joyent
            rackspace

          Configure a New Cloud

          ____________________________________________________________

            localhost
            maas

 QUIT

                          Please press [ENTER] on highlighted Cloud to proceed.
```

Deploy the applications:



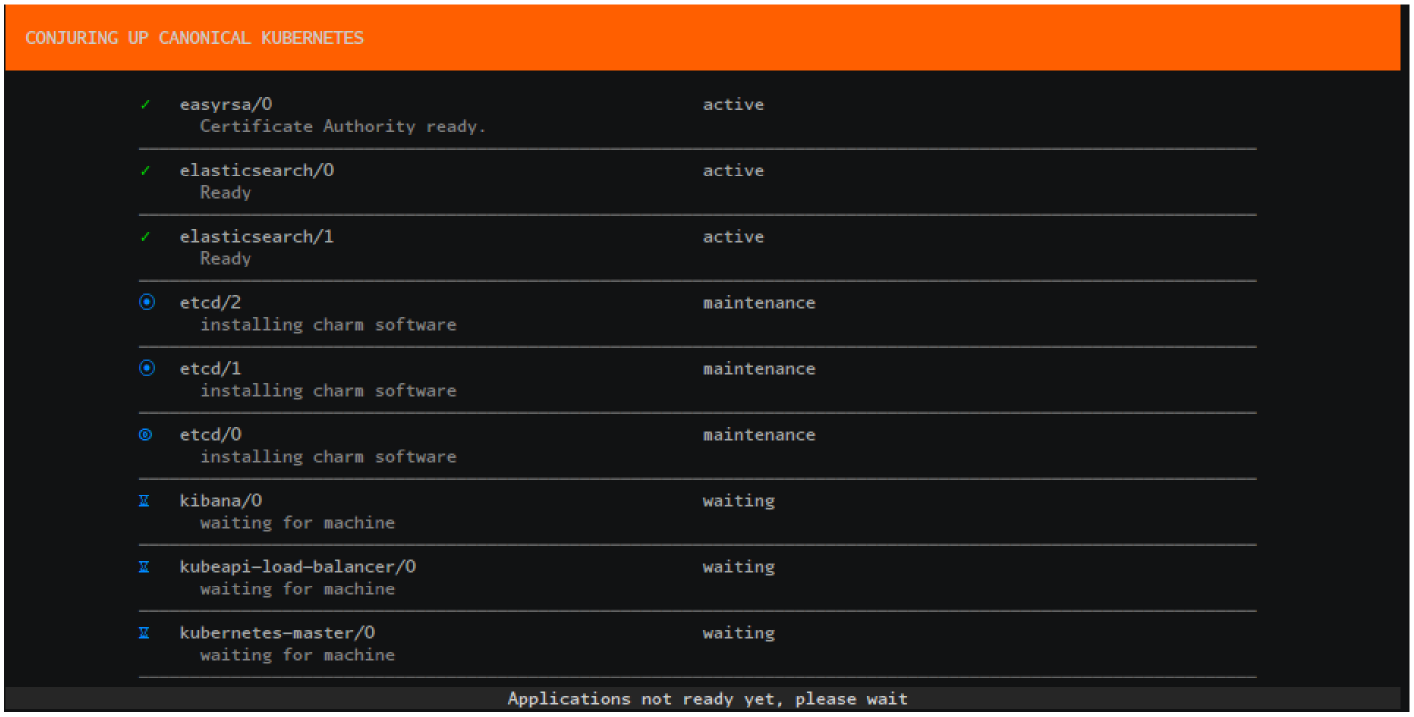
```
REVIEW AND CONFIGURE APPLICATIONS

     10 Applications in kubernetes:

     easyrsa                 Units:   1                       [ Configure        ] [ Deploy          ]

     elasticsearch           Units:   2                       [ Configure        ] [ Deploy          ]

     etcd                    Units:   3                       [ Configure        ] [ Deploy          ]

     filebeat                Units:   0                                            [ Deploy          ]

     flannel                 Units:   0                                            [ Deploy          ]

     kibana                  Units:   1                       [ Configure        ] [ Deploy          ]

     kubeapi-load-balancer   Units:   1                       [ Configure        ] [ Deploy          ]

     This charm delivers the EasyRSA application to act as a Certificate Authority
     (CA) and creates certificates for related charms.

     EasyRSA is a command line utility to build and manage Public Key
     Infrastructure (PKI) Certificate Authority (CA).

                                                      Deploy all 10 Remaining Applications

                          Bootstrapping Juju controller in the background...
```

Wait for Juju bootstrap to finish:



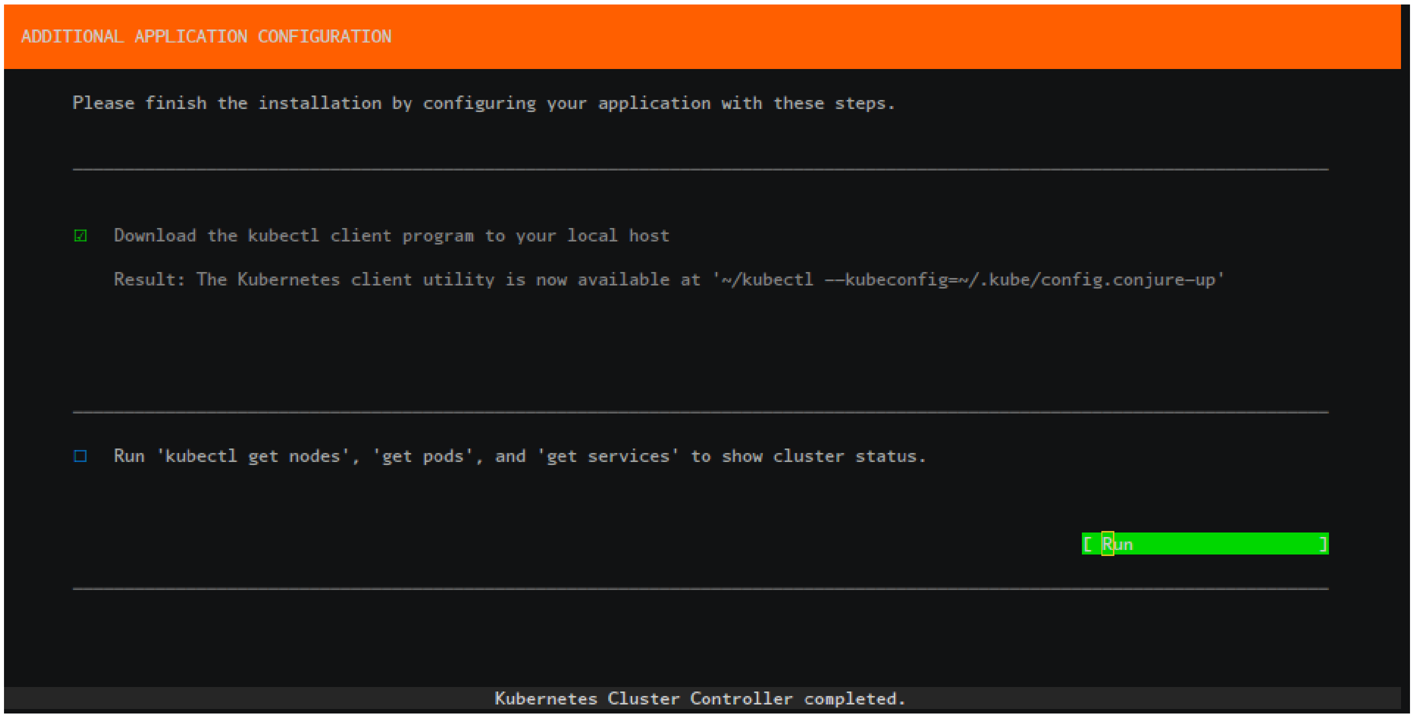
```
WAITING

                         Juju Controller is initializing. Please wait.

                                    - preparing image
                                    - container started

                              - juju-4afc9a-0 (arch=amd64)
                                   Fetching Juju GUI 2.2.3
                                    Waiting for address
                         Attempting to connect to 10.205.94.188:22
                  Logging to /var/log/cloud-init-output.log on the bootstrap machine
                                   Running apt-get update

                         Bootstrapping Juju controller in the background...
```

Wait for our Applications to be fully deployed:



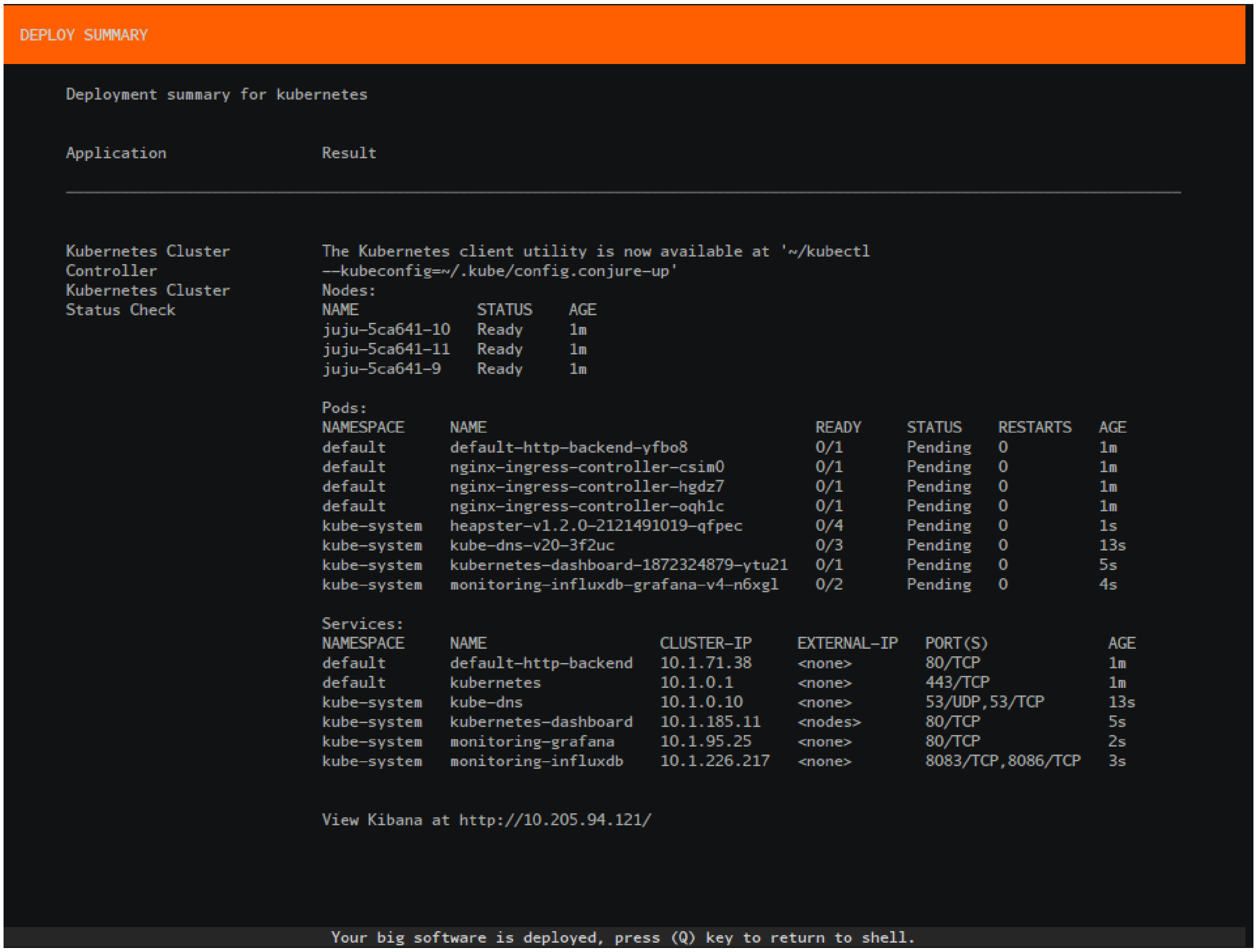
```
CONJURING UP CANONICAL KUBERNETES

  ✓ easyrsa/0                                  active
      Certificate Authority ready.
  ✓ elasticsearch/0                            active
      Ready
  ✓ elasticsearch/1                            active
      Ready
  ⦿ etcd/2                                     maintenance
      installing charm software
  ⦿ etcd/1                                     maintenance
      installing charm software
  ⦿ etcd/0                                     maintenance
      installing charm software
  ⧗ kibana/0                                   waiting
      waiting for machine
  ⧗ kubeapi-load-balancer/0                    waiting
      waiting for machine
  ⧗ kubernetes-master/0                        waiting
      waiting for machine

                        Applications not ready yet, please wait
```

Run the final post-processing steps to automatically configure your Kubernetes environment:



```
ADDITIONAL APPLICATION CONFIGURATION

    Please finish the installation by configuring your application with these steps.

    ☑  Download the kubectl client program to your local host

       Result: The Kubernetes client utility is now available at '~/kubectl --kubeconfig=~/.kube/config.conjure-up'

    ☐  Run 'kubectl get nodes', 'get pods', and 'get services' to show cluster status.

                                                                                          [ Run          ]

                              Kubernetes Cluster Controller completed.
```

Review the final summary screen:



```
DEPLOY SUMMARY

    Deployment summary for kubernetes

    Application                  Result

    Kubernetes Cluster           The Kubernetes client utility is now available at '~/kubectl
    Controller                   --kubeconfig=~/.kube/config.conjure-up'
    Kubernetes Cluster           Nodes:
    Status Check                 NAME              STATUS    AGE
                                 juju-5ca641-10    Ready     1m
                                 juju-5ca641-11    Ready     1m
                                 juju-5ca641-9     Ready     1m

                                 Pods:
                                 NAMESPACE     NAME                                    READY     STATUS    RESTARTS    AGE
                                 default       default-http-backend-yfbo8              0/1       Pending   0           1m
                                 default       nginx-ingress-controller-csim0          0/1       Pending   0           1m
                                 default       nginx-ingress-controller-hgdz7          0/1       Pending   0           1m
                                 default       nginx-ingress-controller-oqh1c          0/1       Pending   0           1m
                                 kube-system   heapster-v1.2.0-2121491019-qfpec        0/4       Pending   0           1s
                                 kube-system   kube-dns-v20-3f2uc                      0/3       Pending   0           13s
                                 kube-system   kubernetes-dashboard-1872324879-ytu21   0/1       Pending   0           5s
                                 kube-system   monitoring-influxdb-grafana-v4-n6xgl    0/2       Pending   0           4s

                                 Services:
                                 NAMESPACE     NAME                      CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
                                 default       default-http-backend      10.1.71.38     <none>        80/TCP              1m
                                 default       kubernetes                10.1.0.1       <none>        443/TCP             1m
                                 kube-system   kube-dns                  10.1.0.10      <none>        53/UDP,53/TCP       13s
                                 kube-system   kubernetes-dashboard      10.1.185.11    <nodes>       80/TCP              5s
                                 kube-system   monitoring-grafana        10.1.95.25     <none>        80/TCP              2s
                                 kube-system   monitoring-influxdb       10.1.226.217   <none>        8083/TCP,8086/TCP   3s


                                 View Kibana at http://10.205.94.121/


                  Your big software is deployed, press (Q) key to return to shell.
```

Accessing the Cluster

You can access your Kubernetes cluster by running the following:

```
kubectl --kubeconfig=~/.kube/config
```

Or if you've already run this once it'll create a new config file as shown in the summary screen.

```
kubectl --kubeconfig=~/.kube/config.conjure-up
```

Logging

Agent Logging

The `juju debug-log` will show all of the consolidated logs of all the Juju agents running on each node of the cluster. This can be useful for finding out why a specific node hasn't deployed or is in an error state. These agent logs are located in `/var/lib/juju/agents` on every node.

See the Juju documentation for more information.

Managing log verbosity

Log verbosity in Juju is set at the model level. You can adjust it at any time:

```
juju add-model k8s-development --config logging-config='<root>=DEBUG;unit=DEBUG'
```

and later

```
juju config-model k8s-production --config logging-config='<root>=ERROR;unit=ERROR'
```

In addition, the jujud daemon is started in debug mode by default on all controllers. To remove that behavior edit `/var/lib/juju/init/jujud-machine-0/exec-start.sh` on the controller node and comment the `--debug` section.

It then contains:

```
#!/usr/bin/env bash

# Set up logging.
touch '/var/log/juju/machine-0.log'
chown syslog:syslog '/var/log/juju/machine-0.log'
chmod 0600 '/var/log/juju/machine-0.log'
exec >> '/var/log/juju/machine-0.log'
exec 2>&1

# Run the script.
'/var/lib/juju/tools/machine-0/jujud' machine --data-dir '/var/lib/juju' --machine
```

Then restart the service with:

```
sudo systemctl restart jujud-machine-0.service
```

See the official documentation for more information about logging and other model settings in Juju.

Windows Server Containers

Kubernetes version 1.5 introduces support for Windows Server Containers. In version 1.5, the Kubernetes control plane (API Server, Scheduler, Controller Manager, etc) continue to run on Linux, while the kubelet and kube-proxy can be run on Windows Server.

Note: Windows Server Containers on Kubernetes is an Alpha feature in Kubernetes 1.5.

Prerequisites

In Kubernetes version 1.5, Windows Server Containers for Kubernetes is supported using the following:

1. Kubernetes control plane running on existing Linux infrastructure (version 1.5 or later).
2. Kubenet network plugin setup on the Linux nodes.
3. Windows Server 2016 (RTM version 10.0.14393 or later).
4. Docker Version 1.12.2-cs2-ws-beta or later for Windows Server nodes (Linux nodes and Kubernetes control plane can run any Kubernetes supported Docker Version).

Networking

Network is achieved using L3 routing. Because third-party networking plugins (e.g. flannel, calico, etc) don’t natively work on Windows Server, existing technology that is built into the Windows and Linux operating systems is relied on. In this L3 networking approach, a /16 subnet is chosen for the cluster nodes, and a /24 subnet is assigned to each worker node. All pods on a given worker node will be connected to the /24 subnet. This allows pods on the same node to communicate with each other. In order to enable networking between pods running on different nodes, routing features that are built into Windows Server 2016 and Linux are used.

Linux

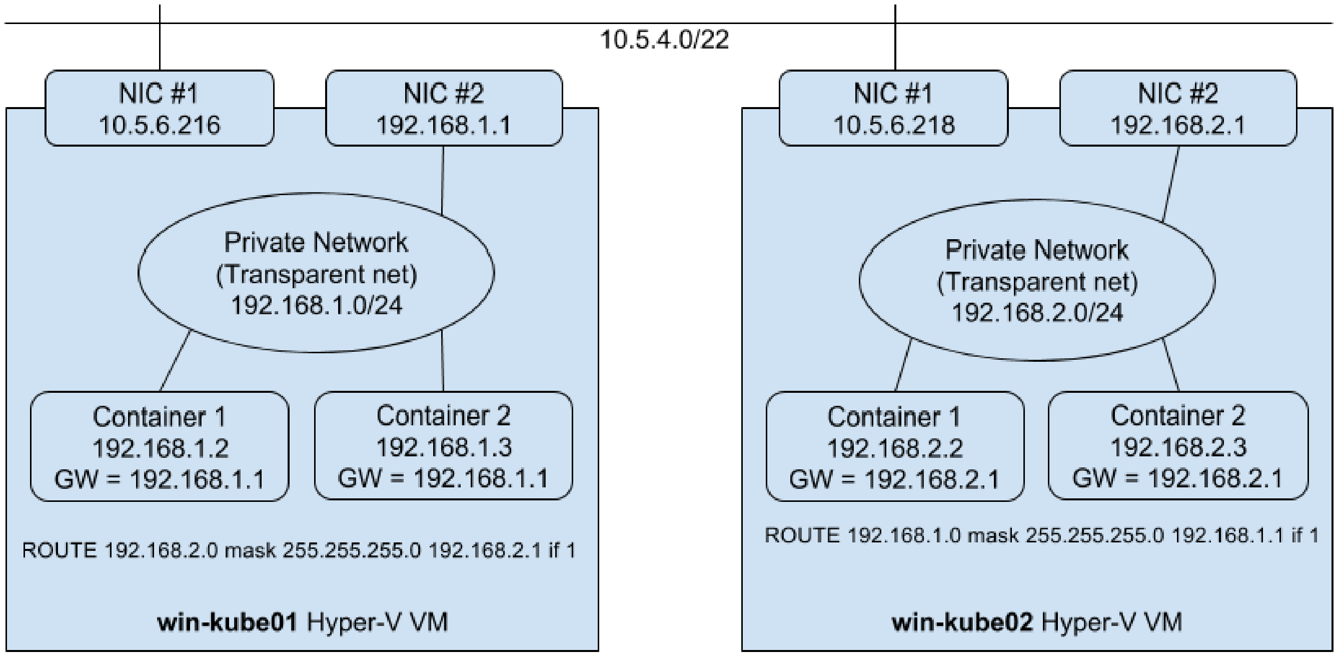
The above networking approach is already supported on Linux using a bridge interface, which essentially creates a private network local to the node. Similar to the Windows side, routes to all other pod CIDRs must be created in order to send packets via the "public" NIC.

Windows

Each Window Server node should have the following configuration:

1. Two NICs (virtual networking adapters) are required on each Windows Server node - The two Windows container networking modes of interest (transparent and L2 bridge) use an external Hyper-V virtual switch. This means that one of the NICs is entirely allocated to the bridge, creating the need for the second NIC.
2. Transparent container network created - This is a manual configuration step and is shown in ***Route Setup*** section below.
3. RRAS (Routing) Windows feature enabled - Allows routing between NICs on the box, and also "captures" packets that have the destination IP of a POD running on the node. To enable, open "Server Manager". Click on "Roles", "Add Roles". Click "Next". Select "Network Policy and Access Services". Click on "Routing and Remote Access Service" and the underlying checkboxes.
4. Routes defined pointing to the other pod CIDRs via the "public" NIC - These routes are added to the built-in routing table as shown in ***Route Setup*** section below.

The following diagram illustrates the Windows Server networking setup for Kubernetes Setup:


10.5.4.0/22
NIC #1 10.5.6.216
NIC #2 192.168.1.1
Private Network (Transparent net) 192.168.1.0/24
Container 1 192.168.1.2 GW = 192.168.1.1
Container 2 192.168.1.3 GW = 192.168.1.1
ROUTE 192.168.2.0 mask 255.255.255.0 192.168.2.1 if 1
win-kube01 Hyper-V VM
NIC #1 10.5.6.218
NIC #2 192.168.2.1
Private Network (Transparent net) 192.168.2.0/24
Container 1 192.168.2.2 GW = 192.168.2.1
Container 2 192.168.2.3 GW = 192.168.2.1
ROUTE 192.168.1.0 mask 255.255.255.0 192.168.1.1 if 1
win-kube02 Hyper-V VM

Setting up Windows Server Containers on Kubernetes

To run Windows Server Containers on Kubernetes, you'll need to set up both your host machines and the Kubernetes node components for Windows and setup Routes for Pod communication on different nodes.

Host Setup

Windows Host Setup

1. Windows Server container host running Windows Server 2016 and Docker v1.12. Follow the setup instructions outlined by this blog post: https://msdn.microsoft.com/en-us/virtualization/windowscontainers/quick_start/quick_start_windows_server.
2. DNS support for Windows recently got merged to docker master and is currently not supported in a stable docker release. To use DNS build docker from master or download the binary from Docker master.
3. Pull the `apprenda/pause` image from `https://hub.docker.com/r/apprenda/pause` .
4. RRAS (Routing) Windows feature enabled.
5. Install a VMSwitch of type `Internal` , by running `New-VMSwitch -Name KubeProxySwitch -SwitchType Internal` command in *PowerShell* window. This will create a new Network Interface with name `vEthernet (KubeProxySwitch)` . This interface will be used by kube-proxy to add Service IPs.

Linux Host Setup

1. Linux hosts should be setup according to their respective distro documentation and the requirements of the Kubernetes version you will be using.
2. CNI network plugin installed.

Component Setup

Requirements

- Git
- Go 1.7.1+
- make (if using Linux or MacOS)
- Important notes and other dependencies are listed here

kubelet

To build the *kubelet*, run:

1. `cd $GOPATH/src/k8s.io/kubernetes`
2. Build *kubelet*
    1. Linux/MacOS: `KUBE_BUILD_PLATFORMS=windows/amd64 make WHAT=cmd/kubelet`
    2. Windows: `go build cmd/kubelet/kubelet.go`

kube-proxy

To build *kube-proxy*, run:

1. `cd $GOPATH/src/k8s.io/kubernetes`
2. Build *kube-proxy*
    1. Linux/MacOS: `KUBE_BUILD_PLATFORMS=windows/amd64 make WHAT=cmd/kube-proxy`
    2. Windows: `go build cmd/kube-proxy/proxy.go`

Route Setup

The below example setup assumes one Linux and two Windows Server 2016 nodes and a cluster CIDR 192.168.0.0/16

Hostname	Routable IP address	Pod CIDR
Lin01	`<IP of Lin01 host>`	192.168.0.0/24
Win01	`<IP of Win01 host>`	192.168.1.0/24

Hostname	Routable IP address	Pod CIDR
Win02	`<IP of Win02 host>`	192.168.2.0/24

Lin01

```
ip route add 192.168.1.0/24 via <IP of Win01 host>
ip route add 192.168.2.0/24 via <IP of Win02 host>
```

Win01

```
docker network create -d transparent --gateway 192.168.1.1 --subnet 192.168.1.0/24
# A bridge is created with Adapter name "vEthernet (HNSTransparent)". Set its IP a
netsh interface ipv4 set address "vEthernet (HNSTransparent)" addr=192.168.1.1
route add 192.168.0.0 mask 255.255.255.0 192.168.0.1 if <Interface Id of the Routa
route add 192.168.2.0 mask 255.255.255.0 192.168.2.1 if <Interface Id of the Routa
```

Win02

```
docker network create -d transparent --gateway 192.168.2.1 --subnet 192.168.2.0/24
# A bridge is created with Adapter name "vEthernet (HNSTransparent)". Set its IP a
netsh interface ipv4 set address "vEthernet (HNSTransparent)" addr=192.168.2.1
route add 192.168.0.0 mask 255.255.255.0 192.168.0.1 if <Interface Id of the Routa
route add 192.168.1.0 mask 255.255.255.0 192.168.1.1 if <Interface Id of the Routa
```

Starting the Cluster

To start your cluster, you'll need to start both the Linux-based Kubernetes control plane, and the Windows Server-based Kubernetes node components. ## Starting the Linux-based Control Plane Use your preferred method to start Kubernetes cluster on Linux. Please note that Cluster CIDR might need to be updated. ## Starting the Windows Node Components To start kubelet on your Windows node: Run the following in a PowerShell window. Be aware that if the node reboots or the process exits, you will have to rerun the commands below to restart the kubelet.

1. Set environment variable *CONTAINER_NETWORK* value to the docker container network to use
   ```
   $env:CONTAINER_NETWORK = "<docker network>"
   ```
2. Run *kubelet* executable using the below command
   ```
   kubelet.exe --hostname-override=<ip address/hostname of the windows node> --pod-infra-container-image="apprenda/pause" --resolv-conf="" --api_servers=<api server location>
   ```

To start kube-proxy on your Windows node:

Run the following in a PowerShell window with administrative privileges. Be aware that if the node reboots or the process exits, you will have to rerun the commands below to restart the kube-proxy.

1. Set environment variable *INTERFACE_TO_ADD_SERVICE_IP* value to `vEthernet (KubeProxySwitch)` which we created in ***Windows Host Setup*** above
   ```
   $env:INTERFACE_TO_ADD_SERVICE_IP = "vEthernet (KubeProxySwitch)"
   ```
2. Run *kube-proxy* executable using the below command
   ```
   .\proxy.exe --v=3 --proxy-mode=userspace --hostname-override=<ip address/hostname of the windows node> --master=<api server location> --bind-address=<ip address of the windows node>
   ```

Scheduling Pods on Windows

Because your cluster has both Linux and Windows nodes, you must explicitly set the nodeSelector constraint to be able to schedule Pods to Windows nodes. You must set nodeSelector with the label beta.kubernetes.io/os to the value windows; see the following example:

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "iis",
    "labels": {
      "name": "iis"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "iis",
        "image": "microsoft/iis",
        "ports": [
          {
            "containerPort": 80
          }
        ]
      }
    ],
    "nodeSelector": {
      "beta.kubernetes.io/os": "windows"
    }
  }
}
```

Known Limitations:

1. There is no network namespace in Windows and as a result currently only one container per pod is supported.
2. Secrets currently do not work because of a bug in Windows Server Containers described here.
3. ConfigMaps have not been implemented yet.
4. `kube-proxy` implementation uses `netsh portproxy` and as it only supports TCP, DNS currently works only if the client retries DNS query using TCP.

Validate Node Setup

Node Conformance Test

Node conformance test is a containerized test framework that provides a system verification and functionality test for a node. The test validates whether the node meets the minimum requirements for Kubernetes; a node that passes the test is qualified to join a Kubernetes cluster.

Limitations

In Kubernetes version 1.5, node conformance test has the following limitations:

- Node conformance test only supports Docker as the container runtime.

Node Prerequisite

To run node conformance test, a node must satisfy the same prerequisites as a standard Kubernetes node. At a minimum, the node should have the following daemons installed:

- Container Runtime (Docker)
- Kubelet

Running Node Conformance Test

To run the node conformance test, perform the following steps:

1. Point your Kubelet to localhost `--api-servers="http://localhost:8080"`, because the test framework starts a local master to test Kubelet. There are some other Kubelet flags you may care:
   1. `--pod-cidr`: If you are using `kubenet`, you should specify an arbitrary CIDR to Kubelet, for example `--pod-cidr=10.180.0.0/24`.
   2. `--cloud-provider`: If you are using `--cloud-provider=gce`, you should remove the flag to run the test.
2. Run the node conformance test with command:

```
# $CONFIG_DIR is the pod manifest path of your Kubelet.
# $LOG_DIR is the test output path.
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  gcr.io/google_containers/node-test:0.2
```

Running Node Conformance Test for Other Architectures

Kubernetes also provides node conformance test docker images for other architectures:

Arch	Image
amd64	node-test-amd64
arm	node-test-arm
arm64	node-test-arm64

Running Selected Test

To run specific tests, overwrite the environment variable `FOCUS` with the regular expression of tests you want to run.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  -e FOCUS=MirrorPod \ # Only run MirrorPod test
  gcr.io/google_containers/node-test:0.2
```

To skip specific tests, overwrite the environment variable `SKIP` with the regular expression of tests you want to skip.

```
sudo docker run -it --rm --privileged --net=host \
  -v /:/rootfs:ro -v $CONFIG_DIR:$CONFIG_DIR -v $LOG_DIR:/var/result \
  -e SKIP=MirrorPod \ # Run all conformance tests but skip MirrorPod test
  gcr.io/google_containers/node-test:0.2
```

Node conformance test is a containerized version of node e2e test. By default, it runs all conformance tests.

Theoretically, you can run any node e2e test if you configure the container and mount required volumes properly. But **it is strongly recommended to only run conformance test**, because it requires much more complex configuration to run non-conformance test.

Caveats

- The test leaves some docker images on the node, including the node conformance test image and images of containers used in the functionality test.
- The test leaves dead containers on the node. These containers are created during the functionality test.

Installing Addons

Overview

Add-ons extend the functionality of Kubernetes.

This page lists some of the available add-ons and links to their respective installation instructions.

Add-ons in each section are sorted alphabetically - the ordering does not imply any preferential status.

Networking and Network Policy

- Calico is a secure L3 networking and network policy provider.
- Canal unites Flannel and Calico, providing networking and network policy.
- Cilium is a L3 network and network policy plugin that can enforce HTTP/API/L7 policies transparently. Both routing and overlay/encapsulation mode are supported.
- Contiv provides configurable networking (native L3 using BGP, overlay using vxlan, classic L2, and Cisco-SDN/ACI) for various use cases and a rich policy framework. Contiv project is fully open sourced. The installer provides both kubeadm and non-kubeadm based installation options.
- Flannel is an overlay network provider that can be used with Kubernetes.
- Romana is a Layer 3 networking solution for pod networks that also supports the NetworkPolicy API. Kubeadm add-on installation details available here.
- Weave Net provides networking and network policy, will carry on working on both sides of a network partition, and does not require an external database.
- CNI-Genie enables Kubernetes to seamlessly connect to a choice of CNI plugins, such as Flannel, Calico, Canal, Romana, or Weave.

Service Discovery

- CoreDNS is a flexible, extensible DNS server which can be installed as the in-cluster DNS for pods.

Visualization & Control

- Dashboard is a dashboard web interface for Kubernetes.
- Weave Scope is a tool for graphically visualizing your containers, pods, services etc. Use it in conjunction with a Weave Cloud account or host the UI yourself.

Legacy Add-ons

There are several other add-ons documented in the deprecated cluster/addons directory.

Well-maintained ones should be linked to here. PRs welcome!

Configuring Kubernetes with Salt

The Kubernetes cluster can be configured using Salt.

The Salt scripts are shared across multiple hosting providers and depending on where you host your Kubernetes cluster, you may be using different operating systems and different networking configurations. As a result, it's important to understand some background information before making Salt changes in order to minimize introducing failures for other hosting providers.

Salt cluster setup

The **salt-master** service runs on the kubernetes-master (except on the default GCE and OpenStack-Heat setup).

The **salt-minion** service runs on the kubernetes-master and each kubernetes-node in the cluster.

Each salt-minion service is configured to interact with the **salt-master** service hosted on the kubernetes-master via the **master.conf** file (except on GCE and OpenStack-Heat).

```
[root@kubernetes-master] $ cat /etc/salt/minion.d/master.conf
master: kubernetes-master
```

The salt-master is contacted by each salt-minion and depending upon the machine information presented, the salt-master will provision the machine as either a kubernetes-master or kubernetes-node with all the required capabilities needed to run Kubernetes.

If you are running the Vagrant based environment, the **salt-api** service is running on the kubernetes-master. It is configured to enable the vagrant user to introspect the salt cluster in order to find out about machines in the Vagrant environment via a REST API.

Standalone Salt Configuration on GCE and others

On GCE and OpenStack, using the Openstack-Heat provider, the master and nodes are all configured as standalone minions. The configuration for each VM is derived from the VM's instance metadata and then stored in Salt grains (`/etc/salt/minion.d/grains.conf`) and pillars (`/srv/salt-overlay/pillar/cluster-params.sls`) that local Salt uses to enforce state.

All remaining sections that refer to master/minion setups should be ignored for GCE and OpenStack. One fallout of this setup is that the Salt mine doesn't exist - there is no sharing of configuration amongst nodes.

Salt security

(Not applicable on default GCE and OpenStack-Heat setup.)

Security is not enabled on the salt-master, and the salt-master is configured to auto-accept incoming requests from minions. It is not recommended to use this security configuration in production environments without deeper study. (In some environments this isn't as bad as it might sound if the salt master port isn't externally accessible and you trust everyone on your network.)

```
[root@kubernetes-master] $ cat /etc/salt/master.d/auto-accept.conf
open_mode: True
auto_accept: True
```

Salt minion configuration

Each minion in the salt cluster has an associated configuration that instructs the salt-master how to provision the required resources on the machine.

An example file is presented below using the Vagrant based environment.

```
[root@kubernetes-master] $ cat /etc/salt/minion.d/grains.conf
grains:
  etcd_servers: $MASTER_IP
  cloud: vagrant
  roles:
    - kubernetes-master
```

Each hosting environment has a slightly different grains.conf file that is used to build conditional logic where required in the Salt files.

The following enumerates the set of defined key/value pairs that are supported today. If you add new ones, please make sure to update this list.

Key	Value
`api_servers`	(Optional) The IP address / host name where a kubelet can get read-only access to kube-apiserver
`cbr-cidr`	(Optional) The minion IP address range used for the docker container bridge.
`cloud`	(Optional) Which IaaS platform is used to host Kubernetes, *gce*, *azure*, *aws*, *vagrant*
`etcd_servers`	(Optional) Comma-delimited list of IP addresses the kube-apiserver and kubelet use to reach etcd. Uses the IP of the first machine in the kubernetes_master role, or 127.0.0.1 on GCE.
`hostnamef`	(Optional) The full host name of the machine, i.e. uname -n
`node_ip`	(Optional) The IP address to use to address this node
`hostname_override`	(Optional) Mapped to the kubelet hostname-override
`network_mode`	(Optional) Networking model to use among nodes: *openvswitch*
`networkInterfaceName`	(Optional) Networking interface to use to bind addresses, default value *eth0*
`publicAddressOverride`	(Optional) The IP address the kube-apiserver should use to bind against for external read-only access
`roles`	(Required) 1. `kubernetes-master` means this machine is the master in the Kubernetes cluster. 2. `kubernetes-pool` means this machine is a kubernetes-node. Depending on the role, the Salt scripts will provision different resources on the machine.

These keys may be leveraged by the Salt sls files to branch behavior.

In addition, a cluster may be running a Debian based operating system or Red Hat based operating system (Centos, Fedora, RHEL, etc.). As a result, it's important to sometimes distinguish behavior based on operating system using if branches like the following.

```
{% if grains['os_family'] == 'RedHat' %}
// something specific to a RedHat environment (Centos, Fedora, RHEL) where you may
{% else %}
// something specific to Debian environment (apt-get, initd)
{% endif %}
```

Best Practices

When configuring default arguments for processes, it’s best to avoid the use of EnvironmentFiles (Systemd in Red Hat environments) or init.d files (Debian distributions) to hold default values that should be common across operating system environments. This helps keep our Salt template files easy to understand for editors who may not be familiar with the particulars of each distribution.

Future enhancements (Networking)

Per pod IP configuration is provider-specific, so when making networking changes, it’s important to sandbox these as all providers may not use the same mechanisms (iptables, openvswitch, etc.)

We should define a grains.conf key that captures more specifically what network configuration environment is being used to avoid future confusion across providers.

Further reading

The cluster/saltbase tree has more details on the current SaltStack configuration.

Building Large Clusters

Support

At v1.8, Kubernetes supports clusters with up to 5000 nodes. More specifically, we support configurations that meet *all* of the following criteria:

- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers
- No more than 100 pods per node

Setup

A cluster is a set of nodes (physical or virtual machines) running Kubernetes agents, managed by a “master” (the cluster-level control plane).

Normally the number of nodes in a cluster is controlled by the value `NUM_NODES` in the platform-specific `config-default.sh` file (for example, see GCE’s `config-default.sh`).

Simply changing that value to something very large, however, may cause the setup script to fail for many cloud providers. A GCE deployment, for example, will run in to quota issues and fail to bring the cluster up.

When setting up a large Kubernetes cluster, the following issues must be considered.

Quota Issues

To avoid running into cloud provider quota issues, when creating a cluster with many nodes, consider:

- Increase the quota for things like CPU, IPs, etc.
  - In GCE, for example, you'll want to increase the quota for:
    - CPUs
    - VM instances
    - Total persistent disk reserved
    - In-use IP addresses
    - Firewall Rules
    - Forwarding rules
    - Routes
    - Target pools
- Gating the setup script so that it brings up new node VMs in smaller batches with waits in between, because some cloud providers rate limit the creation of VMs.

Etcd storage

To improve performance of large clusters, we store events in a separate dedicated etcd instance.

When creating a cluster, existing salt scripts:

- start and configure additional etcd instance
- configure api-server to use it for storing events

Size of master and master components

On GCE/GKE and AWS, `kube-up` automatically configures the proper VM size for your master depending on the number of nodes in your cluster. On other providers, you will need to configure it manually. For reference, the sizes we use on GCE are

- 1-5 nodes: n1-standard-1
- 6-10 nodes: n1-standard-2
- 11-100 nodes: n1-standard-4
- 101-250 nodes: n1-standard-8
- 251-500 nodes: n1-standard-16
- more than 500 nodes: n1-standard-32

And the sizes we use on AWS are

- 1-5 nodes: m3.medium
- 6-10 nodes: m3.large
- 11-100 nodes: m3.xlarge
- 101-250 nodes: m3.2xlarge
- 251-500 nodes: c4.4xlarge
- more than 500 nodes: c4.8xlarge

Note that these master node sizes are currently only set at cluster startup time, and are not adjusted if you later scale your cluster up or down (e.g. manually removing or adding nodes, or using a cluster autoscaler).

Addon Resources

To prevent memory leaks or other resource issues in cluster addons from consuming all the resources available on a node, Kubernetes sets resource limits on addon containers to limit the CPU and Memory resources they can consume (See PR #10653 and #10778).

For example:

```
containers:
- name: fluentd-cloud-logging
  image: gcr.io/google_containers/fluentd-gcp:1.16
  resources:
    limits:
      cpu: 100m
      memory: 200Mi
```

Except for Heapster, these limits are static and are based on data we collected from addons running on 4-node clusters (see #10335). The addons consume a lot more resources when running on large deployment clusters (see #5880). So, if a large cluster is deployed without adjusting these values, the addons may continuously get killed because they keep hitting the limits.

To avoid running into cluster addon resource issues, when creating a cluster with many nodes, consider the following:

- Scale memory and CPU limits for each of the following addons, if used, as you scale up the size of cluster (there is one replica of each handling the entire cluster so memory and CPU usage tends to grow proportionally with size/load on cluster):
  - InfluxDB and Grafana
  - kubedns, dnsmasq, and sidecar
  - Kibana
- Scale number of replicas for the following addons, if used, along with the size of cluster (there are multiple replicas of each so increasing replicas should help handle increased load, but, since load per replica also increases slightly, also consider increasing CPU/memory limits):
  - elasticsearch
- Increase memory and CPU limits slightly for each of the following addons, if used, along with the size of cluster (there is one replica per node but CPU/memory usage increases slightly along with cluster load/size as well):
  - FluentD with ElasticSearch Plugin
  - FluentD with GCP Plugin

Heapster's resource limits are set dynamically based on the initial size of your cluster (see #16185 and #22940). If you find that Heapster is running out of resources, you should adjust the formulas that compute heapster memory request (see those PRs for details).

For directions on how to detect if addon containers are hitting resource limits, see the Troubleshooting section of Compute Resources.

In the future, we anticipate to set all cluster addon resource limits based on cluster size, and to dynamically adjust them if you grow or shrink your cluster. We welcome PRs that implement those features.

Allowing minor node failure at startup

For various reasons (see #18969 for more details) running `kube-up.sh` with a very large `NUM_NODES` may fail due to a very small number of nodes not coming up properly. Currently you have two choices: restart the cluster (`kube-down.sh` and then `kube-up.sh` again), or before running `kube-up.sh` set the environment variable `ALLOWED_NOTREADY_NODES` to whatever value you feel comfortable with. This will allow `kube-up.sh` to succeed with fewer than `NUM_NODES` coming up. Depending on the reason for the failure, those additional nodes may join later or the cluster may remain at a size of `NUM_NODES - ALLOWED_NOTREADY_NODES`.

Running in Multiple Zones

Introduction

Kubernetes 1.2 adds support for running a single cluster in multiple failure zones (GCE calls them simply "zones", AWS calls them "availability zones", here we'll refer to them as "zones"). This is a lightweight version of a broader Cluster Federation feature (previously referred to by the affectionate nickname "Ubernetes"). Full Cluster Federation allows combining separate Kubernetes clusters running in different regions or cloud providers (or on-premises data centers). However, many users simply want to run a more available Kubernetes cluster in multiple zones of their single cloud provider, and this is what the multizone support in 1.2 allows (this previously went by the nickname "Ubernetes Lite").

Multizone support is deliberately limited: a single Kubernetes cluster can run in multiple zones, but only within the same region (and cloud provider). Only GCE and AWS are currently supported automatically (though it is easy to add similar support for other clouds or even bare metal, by simply arranging for the appropriate labels to be added to nodes and volumes).

Functionality

When nodes are started, the kubelet automatically adds labels to them with zone information.

Kubernetes will automatically spread the pods in a replication controller or service across nodes in a single-zone cluster (to reduce the impact of failures.) With multiple-zone clusters, this spreading behaviour is extended across zones (to reduce the impact of zone failures.) (This is achieved via `SelectorSpreadPriority`). This is a best-effort placement, and so if the zones in your cluster are heterogeneous (e.g. different numbers of nodes, different types of nodes, or different pod resource requirements), this might prevent perfectly even spreading of your pods across zones. If desired, you can use homogenous zones (same number and types of nodes) to reduce the probability of unequal spreading.

When persistent volumes are created, the `PersistentVolumeLabel` admission controller automatically adds zone labels to them. The scheduler (via the `VolumeZonePredicate` predicate) will then ensure that pods that claim a given volume are only placed into the same zone as that volume, as volumes cannot be attached across zones.

Limitations

There are some important limitations of the multizone support:

- We assume that the different zones are located close to each other in the network, so we don't perform any zone-aware routing. In particular, traffic that goes via services might cross zones (even if pods in some pods backing that service exist in the same zone as the client), and this may incur additional latency and cost.
- Volume zone-affinity will only work with a `PersistentVolume`, and will not work if you directly specify an EBS volume in the pod spec (for example).
- Clusters cannot span clouds or regions (this functionality will require full federation support).
- Although your nodes are in multiple zones, kube-up currently builds a single master node by default. While services are highly available and can tolerate the loss of a zone, the control plane is located in a single zone. Users that want a highly available control plane should follow the high availability instructions.
- StatefulSet volume zone spreading when using dynamic provisioning is currently not compatible with pod affinity or anti-affinity policies.

- If the name of the StatefulSet contains dashes (“-“), volume zone spreading may not provide a uniform distribution of storage across zones.
- When specifying multiple PVCs in a Deployment or Pod spec, the StorageClass needs to be configured for a specific, single zone, or the PVs need to be statically provisioned in a specific zone. Another workaround is to use a StatefulSet, which will ensure that all the volumes for a replica are provisioned in the same zone.

Walkthrough

We’re now going to walk through setting up and using a multi-zone cluster on both GCE & AWS. To do so, you bring up a full cluster (specifying `MULTIZONE=true`), and then you add nodes in additional zones by running `kube-up` again (specifying `KUBE_USE_EXISTING_MASTER=true`).

Bringing up your cluster

Create the cluster as normal, but pass MULTIZONE to tell the cluster to manage multiple zones; creating nodes in us-central1-a.

GCE:

```
curl -sS https://get.k8s.io | MULTIZONE=true KUBERNETES_PROVIDER=gce KUBE_GCE_ZONE
```

AWS:

```
curl -sS https://get.k8s.io | MULTIZONE=true KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE
```

This step brings up a cluster as normal, still running in a single zone (but `MULTIZONE=true` has enabled multi-zone capabilities).

Nodes are labeled

View the nodes; you can see that they are labeled with zone information. They are all in `us-central1-a` (GCE) or `us-west-2a` (AWS) so far. The labels are `failure-domain.beta.kubernetes.io/region` for the region, and `failure-domain.beta.kubernetes.io/zone` for the zone:

```
> kubectl get nodes --show-labels


NAME                     STATUS                     AGE    VERSION          LABELS
kubernetes-master        Ready,SchedulingDisabled   6m     v1.6.0+fff5156   beta.ku
kubernetes-minion-87j9   Ready                      6m     v1.6.0+fff5156   beta.ku
kubernetes-minion-9vlv   Ready                      6m     v1.6.0+fff5156   beta.ku
kubernetes-minion-a12q   Ready                      6m     v1.6.0+fff5156   beta.ku
```

Add more nodes in a second zone

Let's add another set of nodes to the existing cluster, reusing the existing master, running in a different zone (us-central1-b or us-west-2b). We run kube-up again, but by specifying `KUBE_USE_EXISTING_MASTER=true` kube-up will not create a new master, but will reuse one that was previously created instead.

GCE:

```
KUBE_USE_EXISTING_MASTER=true MULTIZONE=true KUBERNETES_PROVIDER=gce KUBE_GCE_ZONE
```

On AWS we also need to specify the network CIDR for the additional subnet, along with the master internal IP address:

```
KUBE_USE_EXISTING_MASTER=true MULTIZONE=true KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE
```

View the nodes again; 3 more nodes should have launched and be tagged in us-central1-b:

```
> kubectl get nodes --show-labels

NAME                     STATUS                     AGE   VERSION          LABELS
kubernetes-master        Ready,SchedulingDisabled   16m   v1.6.0+fff5156   beta.k
kubernetes-minion-281d   Ready                      2m    v1.6.0+fff5156   beta.k
kubernetes-minion-87j9   Ready                      16m   v1.6.0+fff5156   beta.k
kubernetes-minion-9vlv   Ready                      16m   v1.6.0+fff5156   beta.k
kubernetes-minion-a12q   Ready                      17m   v1.6.0+fff5156   beta.k
kubernetes-minion-pp2f   Ready                      2m    v1.6.0+fff5156   beta.k
kubernetes-minion-wf8i   Ready                      2m    v1.6.0+fff5156   beta.k
```

Volume affinity

Create a volume using the dynamic volume creation (only PersistentVolumes are supported for zone affinity):

```
kubectl create -f - <<EOF
{
  "kind": "PersistentVolumeClaim",
  "apiVersion": "v1",
  "metadata": {
    "name": "claim1",
    "annotations": {
        "volume.alpha.kubernetes.io/storage-class": "foo"
    }
  },
  "spec": {
    "accessModes": [
      "ReadWriteOnce"
    ],
    "resources": {
      "requests": {
        "storage": "5Gi"
      }
    }
  }
}
EOF
```

NOTE: For version 1.3+ Kubernetes will distribute dynamic PV claims across the configured zones. For version 1.2, dynamic persistent volumes were always created in the zone of the cluster master (here us-central1-a / us-west-2a); that issue (#23330) was addressed in 1.3+.

Now lets validate that Kubernetes automatically labeled the zone & region the PV was created in.

```
> kubectl get pv --show-labels
NAME           CAPACITY   ACCESSMODES   STATUS    CLAIM            REASON    AGE
pv-gce-mj4gm   5Gi        RWO           Bound     default/claim1             46s
```

So now we will create a pod that uses the persistent volume claim. Because GCE PDs / AWS EBS volumes cannot be attached across zones, this means that this pod can only be created in the same zone as the volume:

```
kubectl create -f - <<EOF
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: claim1
EOF
```

Note that the pod was automatically created in the same zone as the volume, as cross-zone attachments are not generally permitted by cloud providers:

```
> kubectl describe pod mypod | grep Node
Node:        kubernetes-minion-9vlv/10.240.0.5
> kubectl get node kubernetes-minion-9vlv --show-labels
NAME                     STATUS    AGE    VERSION          LABELS
kubernetes-minion-9vlv   Ready     22m    v1.6.0+fff5156   beta.kubernetes.io/inst
```

Pods are spread across zones

Pods in a replication controller or service are automatically spread across zones. First, let's launch more nodes in a third zone:

GCE:

```
KUBE_USE_EXISTING_MASTER=true MULTIZONE=true KUBERNETES_PROVIDER=gce KUBE_GCE_ZONE
```

AWS:

```
KUBE_USE_EXISTING_MASTER=true MULTIZONE=true KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE
```

Verify that you now have nodes in 3 zones:

```
kubectl get nodes --show-labels
```

Create the guestbook-go example, which includes an RC of size 3, running a simple web app:

```
find kubernetes/examples/guestbook-go/ -name '*.json' | xargs -I {} kubectl create
```

The pods should be spread across all 3 zones:

```
>  kubectl describe pod -l app=guestbook | grep Node
Node:        kubernetes-minion-9vlv/10.240.0.5
Node:        kubernetes-minion-281d/10.240.0.8
Node:        kubernetes-minion-olsh/10.240.0.11

 > kubectl get node kubernetes-minion-9vlv kubernetes-minion-281d kubernetes-minio
NAME                     STATUS    AGE    VERSION          LABELS
kubernetes-minion-9vlv   Ready     34m    v1.6.0+fff5156   beta.kubernetes.io/inst
kubernetes-minion-281d   Ready     20m    v1.6.0+fff5156   beta.kubernetes.io/inst
kubernetes-minion-olsh   Ready     3m     v1.6.0+fff5156   beta.kubernetes.io/inst
```

Load-balancers span all zones in a cluster; the guestbook-go example includes an example load-balanced service:

```
> kubectl describe service guestbook | grep LoadBalancer.Ingress
LoadBalancer Ingress:   130.211.126.21

> ip=130.211.126.21

> curl -s http://${ip}:3000/env | grep HOSTNAME
  "HOSTNAME": "guestbook-44sep",

> (for i in `seq 20`; do curl -s http://${ip}:3000/env | grep HOSTNAME; done)  | s
  "HOSTNAME": "guestbook-44sep",
  "HOSTNAME": "guestbook-hum5n",
  "HOSTNAME": "guestbook-ppm40",
```

The load balancer correctly targets all the pods, even though they are in multiple zones.

Shutting down the cluster

When you're done, clean up:

GCE:

```
KUBERNETES_PROVIDER=gce KUBE_USE_EXISTING_MASTER=true KUBE_GCE_ZONE=us-central1-f
KUBERNETES_PROVIDER=gce KUBE_USE_EXISTING_MASTER=true KUBE_GCE_ZONE=us-central1-b
KUBERNETES_PROVIDER=gce KUBE_GCE_ZONE=us-central1-a kubernetes/cluster/kube-down.s
```

AWS:

```
KUBERNETES_PROVIDER=aws KUBE_USE_EXISTING_MASTER=true KUBE_AWS_ZONE=us-west-2c kub
KUBERNETES_PROVIDER=aws KUBE_USE_EXISTING_MASTER=true KUBE_AWS_ZONE=us-west-2b kub
KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE=us-west-2a kubernetes/cluster/kube-down.sh
```

Building High-Availability Clusters

Introduction

This document describes how to build a high-availability (HA) Kubernetes cluster. This is a fairly advanced topic. Users who merely want to experiment with Kubernetes are encouraged to use configurations that are simpler to set up such as Minikube or try Google Container Engine for hosted Kubernetes.

Also, at this time high availability support for Kubernetes is not continuously tested in our end-to-end (e2e) testing. We will be working to add this continuous testing, but for now the single-node master installations are more heavily tested.

Overview

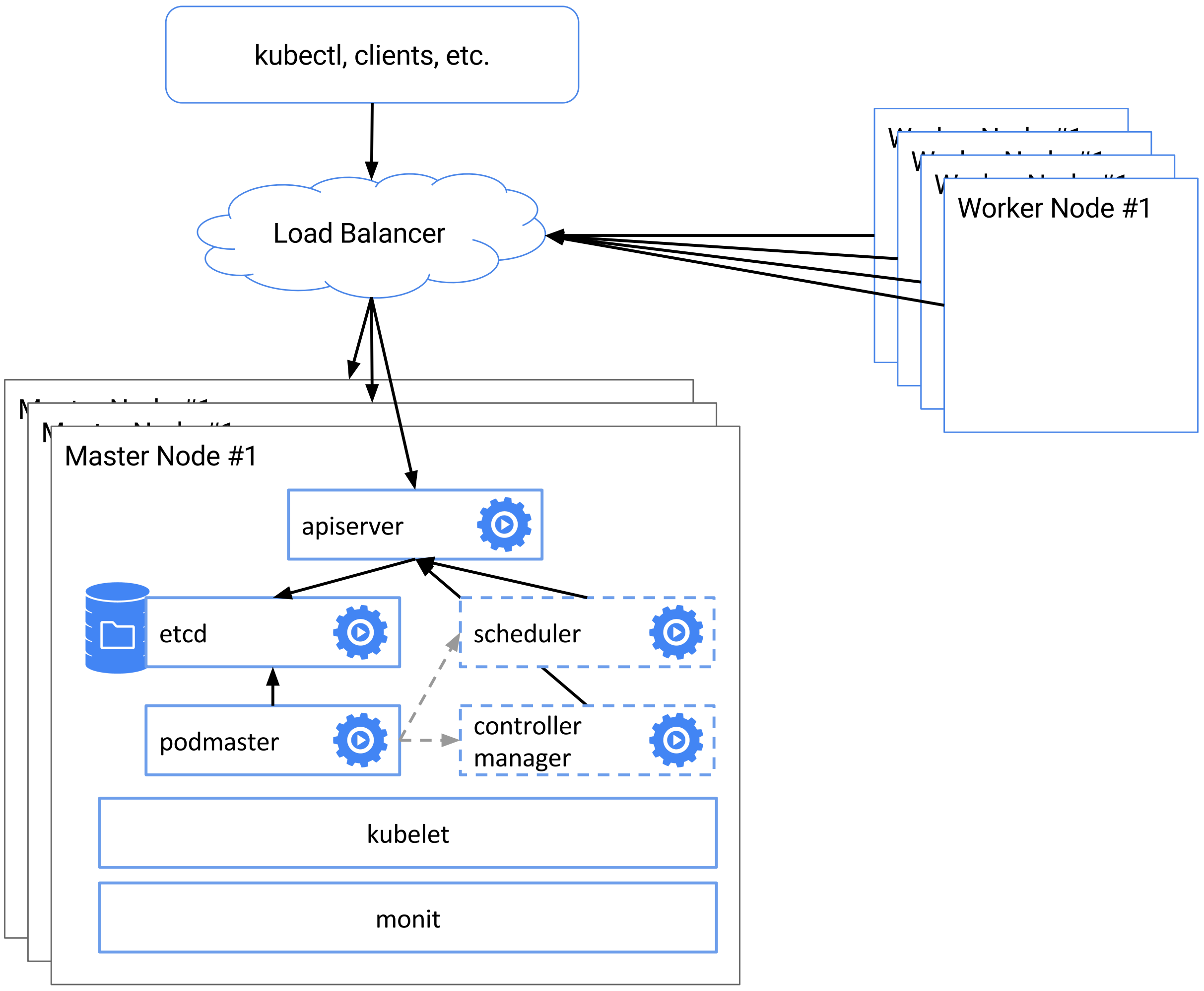
Setting up a truly reliable, highly available distributed system requires a number of steps. It is akin to wearing underwear, pants, a belt, suspenders, another pair of underwear, and another pair of pants.

We go into each of these steps in detail, but a summary is given here to help guide and orient the user.

The steps involved are as follows:

- Creating the reliable constituent nodes that collectively form our HA master implementation.
- Setting up a redundant, reliable storage layer with clustered etcd.
- Starting replicated, load balanced Kubernetes API servers
- Setting up master-elected Kubernetes scheduler and controller-manager daemons

Here's what the system should look like when it's finished:



Initial set-up

The remainder of this guide assumes that you are setting up a 3-node clustered master, where each machine is running some flavor of Linux. Examples in the guide are given for Debian distributions, but they should be easily adaptable to other distributions. Likewise, this set up should work whether you are running in a public or private cloud provider, or if you are running on bare metal.

The easiest way to implement an HA Kubernetes cluster is to start with an existing single-master cluster. The instructions at https://get.k8s.io describe easy installation for single-master clusters on a variety of platforms.

Reliable nodes

On each master node, we are going to run a number of processes that implement the Kubernetes API. The first step in making these reliable is to make sure that each automatically restarts when it fails. To achieve this, we need to install a process watcher. We choose to use the `kubelet` that we run on each of the worker nodes. This is convenient, since we can use containers to distribute our binaries, we can establish resource limits, and introspect the resource usage of each daemon. Of course, we also need something to monitor the kubelet itself (insert who watches the watcher jokes here). For Debian systems, we choose monit, but there are a number of alternate choices. For example, on systemd-based systems (e.g. RHEL, CentOS), you can run 'systemctl enable kubelet'.

If you are extending from a standard Kubernetes installation, the `kubelet` binary should already be present on your system. You can run `which kubelet` to determine if the binary is in fact installed. If it is not installed, you should install the kubelet binary, the kubelet init file and default-kubelet scripts.

If you are using monit, you should also install the monit daemon (`apt-get install monit`) and the monit-kubelet and monit-docker configs.

On systemd systems you `systemctl enable kubelet` and `systemctl enable docker`.

Establishing a redundant, reliable data storage layer

The central foundation of a highly available solution is a redundant, reliable storage layer. The number one rule of high-availability is to protect the data. Whatever else happens, whatever catches on fire, if you have the data, you can rebuild. If you lose the data, you're done.

Clustered etcd already replicates your storage to all master instances in your cluster. This means that to lose data, all three nodes would need to have their physical (or virtual) disks fail at the same time. The probability that this occurs is relatively low, so for many people running a replicated etcd cluster is likely reliable enough. You can add additional reliability by increasing the size of the cluster from three to five nodes. If that is still insufficient, you can add even more redundancy to your storage layer.

Clustering etcd

The full details of clustering etcd are beyond the scope of this document, lots of details are given on the etcd clustering page. This example walks through a simple cluster set up, using etcd's built in discovery to build our cluster.

First, hit the etcd discovery service to create a new token:

```
curl https://discovery.etcd.io/new?size=3
```

On each node, copy the etcd.yaml file into `/etc/kubernetes/manifests/etcd.yaml`

The kubelet on each node actively monitors the contents of that directory, and it will create an instance of the `etcd` server from the definition of the pod specified in `etcd.yaml`.

Note that in `etcd.yaml` you should substitute the token URL you got above for `${DISCOVERY_TOKEN}` on all three machines, and you should substitute a different name (e.g. `node-1`) for `${NODE_NAME}` and the correct IP address for `${NODE_IP}` on each machine.

Validating your cluster

Once you copy this into all three nodes, you should have a clustered etcd set up. You can validate on master with

```
kubectl exec < pod_name > etcdctl member list
```

and

```
kubectl exec < pod_name > etcdctl cluster-health
```

You can also validate that this is working with `etcdctl set foo bar` on one node, and `etcdctl get foo` on a different node.

Even more reliable storage

Of course, if you are interested in increased data reliability, there are further options which make the place where etcd installs its data even more reliable than regular disks (belts *and* suspenders, ftw!).

If you use a cloud provider, then they usually provide this for you, for example Persistent Disk on the Google Cloud Platform. These are block-device persistent storage that can be mounted onto your virtual machine. Other cloud providers provide similar solutions.

If you are running on physical machines, you can also use network attached redundant storage using an iSCSI or NFS interface. Alternatively, you can run a clustered file system like Gluster or Ceph. Finally, you can also run a RAID array on each physical machine.

Regardless of how you choose to implement it, if you chose to use one of these options, you should make sure that your storage is mounted to each machine. If your storage is shared between the three masters in your cluster, you should create a different directory on the storage for each node. Throughout these instructions, we assume that this storage is mounted to your machine in `/var/etcd/data`.

Replicated API Servers

Once you have replicated etcd set up correctly, we will also install the apiserver using the kubelet.

Installing configuration files

First you need to create the initial log file, so that Docker mounts a file instead of a directory:

```
touch /var/log/kube-apiserver.log
```

Next, you need to create a `/srv/kubernetes/` directory on each node. This directory includes:

- basic_auth.csv - basic auth user and password
- ca.crt - Certificate Authority cert

- known_tokens.csv - tokens that entities (e.g. the kubelet) can use to talk to the apiserver
- kubecfg.crt - Client certificate, public key
- kubecfg.key - Client certificate, private key
- server.cert - Server certificate, public key
- server.key - Server certificate, private key

The easiest way to create this directory, may be to copy it from the master node of a working cluster, or you can manually generate these files yourself.

Starting the API Server

Once these files exist, copy the kube-apiserver.yaml into `/etc/kubernetes/manifests/` on each master node.

The kubelet monitors this directory, and will automatically create an instance of the `kube-apiserver` container using the pod definition specified in the file.

Load balancing

At this point, you should have 3 apiservers all working correctly. If you set up a network load balancer, you should be able to access your cluster via that load balancer, and see traffic balancing between the apiserver instances. Setting up a load balancer will depend on the specifics of your platform, for example instructions for the Google Cloud Platform can be found here.

Note, if you are using authentication, you may need to regenerate your certificate to include the IP address of the balancer, in addition to the IP addresses of the individual nodes.

For pods that you deploy into the cluster, the `kubernetes` service/dns name should provide a load balanced endpoint for the master automatically.

For external users of the API (e.g. the `kubectl` command line interface, continuous build pipelines, or other clients) you will want to configure them to talk to the external load balancer's IP address.

Master elected components

So far we have set up state storage, and we have set up the API server, but we haven't run anything that actually modifies cluster state, such as the controller manager and scheduler. To achieve this reliably, we only want to have one actor modifying state at a time, but we want replicated instances of these actors, in case a machine dies. To achieve this, we are going to use a lease-lock in the API to perform master election. We will use the `--leader-elect` flag for each scheduler and controller-manager, using a lease in the API will ensure that only 1 instance of the scheduler and controller-manager are running at once.

The scheduler and controller-manager can be configured to talk to the API server that is on the same node (i.e. 127.0.0.1), or it can be configured to communicate using the load balanced IP address of the API servers. Regardless of how they are configured, the scheduler and controller-manager will complete the leader election process mentioned above when using the `--leader-elect` flag.

In case of a failure accessing the API server, the elected leader will not be able to renew the lease, causing a new leader to be elected. This is especially relevant when configuring the scheduler and controller-manager to access the API server via 127.0.0.1, and the API server on the same node is unavailable.

Installing configuration files

First, create empty log files on each node, so that Docker will mount the files not make new directories:

```
touch /var/log/kube-scheduler.log
touch /var/log/kube-controller-manager.log
```

Next, set up the descriptions of the scheduler and controller manager pods on each node by copying kube-scheduler.yaml and kube-controller-manager.yaml into the `/etc/kubernetes/manifests/` directory.

Conclusion

At this point, you are done (yeah!) with the master components, but you still need to add worker nodes (boo!).

If you have an existing cluster, this is as simple as reconfiguring your kubelets to talk to the load-balanced endpoint, and restarting the kubelets on each node.

If you are turning up a fresh cluster, you will need to install the kubelet and kube-proxy on each worker node, and set the `--apiserver` flag to your replicated endpoint.

Downloading or Building Kubernetes

You can either build a release from sources or download a pre-built release. If you do not plan on developing Kubernetes itself, we suggest a pre-built release.

If you just want to run Kubernetes locally for development, we recommend using Minikube. You can download Minikube here. Minikube sets up a local VM that runs a Kubernetes cluster securely, and makes it easy to work with that cluster.

Prebuilt Binary Release

The list of binary releases is available for download from the GitHub Kubernetes repo release page.

Download the latest release and unpack this tar file on Linux or OS X, cd to the created `kubernetes/` directory, and then follow the getting started guide for your cloud.

On OS X you can also use the homebrew package manager: `brew install kubernetes-cli`

Building from source

Get the Kubernetes source. If you are simply building a release from source there is no need to set up a full golang environment as all building happens in a Docker container.

Building a release is simple.

```
git clone https://github.com/kubernetes/kubernetes.git
cd kubernetes
make release
```

For more details on the release process see the `build` directory

Download Kubernetes and automatically set up a default cluster

The bash script at `https://get.k8s.io`, which can be run with `wget` or `curl`, automatically downloads Kubernetes, and provisions a cluster based on your desired cloud provider.

```
# wget version
export KUBERNETES_PROVIDER=YOUR_PROVIDER; wget -q -O - https://get.k8s.io | bash

# curl version
export KUBERNETES_PROVIDER=YOUR_PROVIDER; curl -sS https://get.k8s.io | bash
```

Possible values for `YOUR_PROVIDER` include:

- `gce` - Google Compute Engine [default]
- `gke` - Google Container Engine
- `aws` - Amazon EC2
- `azure` - Microsoft Azure
- `vagrant` - Vagrant (on local virtual machines)
- `vsphere` - VMWare VSphere
- `rackspace` - Rackspace

For the complete, up-to-date list of providers supported by this script, see the `/cluster` folder in the main Kubernetes repo, where each folder represents a possible value for `YOUR_PROVIDER`. If you don't see your desired provider, try looking at our getting started guides; there's a good chance we have docs for them.

Concepts

The Concepts section helps you learn about the parts of the Kubernetes system and the abstractions Kubernetes uses to represent your cluster, and helps you obtain a deeper understanding of how Kubernetes works.

Overview

To work with Kubernetes, you use *Kubernetes API objects* to describe your cluster's *desired state*: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. You set your desired state by creating objects using the Kubernetes API, typically via the command-line interface, `kubectl`. You can also use the Kubernetes API directly to interact with the cluster and set or modify your desired state.

Once you've set your desired state, the *Kubernetes Control Plane* works to make the cluster's current state match the desired state. To do so, Kubernetes performs a variety of tasks automatically–such as starting or restarting containers, scaling the number of replicas of a given application, and more. The Kubernetes Control Plane consists of a collection of processes running on your cluster:

- The **Kubernetes Master** is a collection of three processes that run on a single node in your cluster, which is designated as the master node. Those processes are: kube-apiserver, kube-controller-manager and kube-scheduler.
- Each individual non-master node in your cluster runs two processes:
  - **kubelet**, which communicates with the Kubernetes Master.
  - **kube-proxy**, a network proxy which reflects Kubernetes networking services on each node.

Kubernetes Objects

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing. These abstractions are represented by objects in the Kubernetes API; see the Kubernetes Objects overview for more details.

The basic Kubernetes objects include:

- Pod
- Service
- Volume
- Namespace

In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality and convenience features. They include:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

Kubernetes Control Plane

The various parts of the Kubernetes Control Plane, such as the Kubernetes Master and kubelet processes, govern how Kubernetes communicates with your cluster. The Control Plane maintains a record of all of the Kubernetes Objects in the system, and runs continuous control loops to manage those objects' state. At any given time, the Control Plane's control loops will respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you provided.

For example, when you use the Kubernetes API to create a Deployment object, you provide a new desired state for the system. The Kubernetes Control Plane records that object creation, and carries out your instructions by starting the required applications and scheduling them to cluster nodes–thus making the cluster's actual state match the desired state.

Kubernetes Master

The Kubernetes master is responsible for maintaining the desired state for your cluster. When you interact with Kubernetes, such as by using the `kubectl` command-line interface, you're communicating with your cluster's Kubernetes master.
The "master" refers to a collection of processes managing the cluster state. Typically these processes are all run on a single node in the cluster, and this node is also referred to as the master. The master can also be replicated for availability and redundancy.

Kubernetes Nodes

The nodes in a cluster are the machines (VMs, physical servers, etc) that run your applications and cloud workflows. The Kubernetes master controls each node; you'll rarely interact with nodes directly.

Object Metadata

- Annotations

What's next

If you would like to write a concept page, see Using Page Templates for information about the concept page type and the concept template.

What is Kubernetes?

This page is an overview of Kubernetes.

Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers.

With Kubernetes, you are able to quickly and efficiently respond to customer demand:

- Deploy your applications quickly and predictably.
- Scale your applications on the fly.
- Roll out new features seamlessly.
- Limit hardware usage to required resources only.

Our goal is to foster an ecosystem of components and tools that relieve the burden of running applications in public and private clouds.

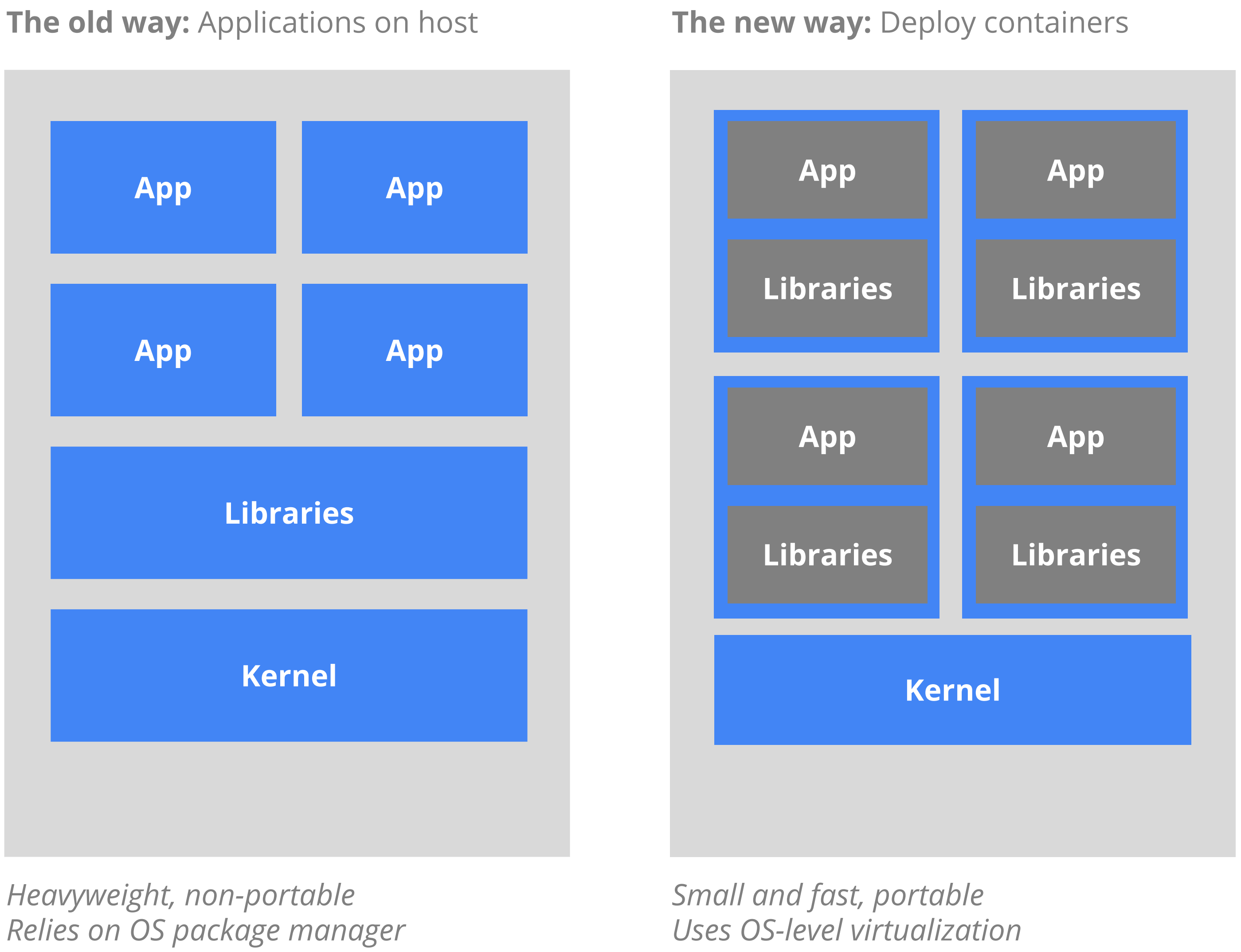
Kubernetes is

- **Portable**: public, private, hybrid, multi-cloud
- **Extensible**: modular, pluggable, hookable, composable
- **Self-healing**: auto-placement, auto-restart, auto-replication, auto-scaling

Google started the Kubernetes project in 2014. Kubernetes builds upon a decade and a half of experience that Google has with running production workloads at scale, combined with best-of-breed ideas and practices from the community.

Why containers?

Looking for reasons why you should be using containers?


The old way: Applications on host
App
App
App
App
Libraries
Kernel
Heavyweight, non-portable
Relies on OS package manager
The new way: Deploy containers
App
Libraries
App
Libraries
App
Libraries
App
Libraries
Kernel
Small and fast, portable
Uses OS-level virtualization

The *Old Way* to deploy applications was to install the applications on a host using the operating system package manager. This had the disadvantage of entangling the applications' executables, configuration, libraries, and lifecycles with each other and with the host OS. One could build immutable virtual-machine images in order to achieve predictable rollouts and rollbacks, but VMs are heavyweight and non-portable.

The *New Way* is to deploy containers based on operating-system-level virtualization rather than hardware virtualization. These containers are isolated from each other and from the host: they have their own filesystems, they can't see each others' processes, and their computational resource usage can be bounded. They are easier to build than VMs, and because they are decoupled from the underlying infrastructure and from the host filesystem, they are portable across clouds and OS distributions.

Because containers are small and fast, one application can be packed in each container image. This one-to-one application-to-image relationship unlocks the full benefits of containers. With containers, immutable container images can be created at build/release time rather than deployment time, since each application doesn't need to be composed with the rest of the application stack, nor married to the production infrastructure environment. Generating container images at build/release time enables a consistent environment to be carried from development into production. Similarly, containers are vastly more transparent than VMs, which facilitates monitoring and management. This is especially true when the containers' process lifecycles are managed by the infrastructure rather than hidden by a process supervisor inside the container. Finally, with a single application per container, managing the containers becomes tantamount to managing deployment of the application.

Summary of container benefits:

- **Agile application creation and deployment**: Increased ease and efficiency of container image creation compared to VM image use.
- **Continuous development, integration, and deployment**: Provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- **Dev and Ops separation of concerns**: Create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- **Environmental consistency across development, testing, and production**: Runs the same on a laptop as it does in the cloud.
- **Cloud and OS distribution portability**: Runs on Ubuntu, RHEL, CoreOS, on-prem, Google Container Engine, and anywhere else.
- **Application-centric management**: Raises the level of abstraction from running an OS on virtual hardware to run an application on an OS using logical resources.
- **Loosely coupled, distributed, elastic, liberated micro-services**: Applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a fat monolithic stack running on one big single-purpose machine.

- **Resource isolation**: Predictable application performance.
- **Resource utilization**: High efficiency and density.

Why do I need Kubernetes and what can it do?

At a minimum, Kubernetes can schedule and run application containers on clusters of physical or virtual machines. However, Kubernetes also allows developers to 'cut the cord' to physical and virtual machines, moving from a **host-centric** infrastructure to a **container-centric** infrastructure, which provides the full advantages and benefits inherent to containers. Kubernetes provides the infrastructure to build a truly **container-centric** development environment.

Kubernetes satisfies a number of common needs of applications running in production, such as:

- Co-locating helper processes, facilitating composite applications and preserving the one-application-per-container model
- Mounting storage systems
- Distributing secrets
- Checking application health
- Replicating application instances
- Using Horizontal Pod Autoscaling
- Naming and discovering
- Balancing loads
- Rolling updates
- Monitoring resources
- Accessing and ingesting logs
- Debugging applications
- Providing authentication and authorization

This provides the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and facilitates portability across infrastructure providers.

How is Kubernetes a platform?

Even though Kubernetes provides a lot of functionality, there are always new scenarios that would benefit from new features. Application-specific workflows can be streamlined to accelerate developer velocity. Ad hoc orchestration that is acceptable initially often requires robust automation at scale. This is why Kubernetes was also designed to serve as a platform for building an ecosystem of components and tools to make it easier to deploy, scale, and manage applications.

Labels empower users to organize their resources however they please. Annotations enable users to decorate resources with custom information to facilitate their workflows and provide an easy way for management tools to checkpoint state.

Additionally, the Kubernetes control plane is built upon the same APIs that are available to developers and users. Users can write their own controllers, such as schedulers, with their own APIs that can be targeted by a general-purpose command-line tool.

This design has enabled a number of other systems to build atop Kubernetes.

What Kubernetes is not

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. It preserves user choice where it is important.

Kubernetes:

- Does not limit the types of applications supported. It does not dictate application frameworks (e.g., Wildfly), restrict the set of supported language runtimes (for example, Java, Python, Ruby), cater to only 12-factor applications, nor distinguish *apps* from *services*. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not provide middleware (e.g., message buses), data-processing frameworks (for example, Spark), databases (e.g., mysql), nor cluster storage systems (e.g., Ceph) as built-in services. Such applications run on Kubernetes.
- Does not have a click-to-deploy service marketplace.
- Does not deploy source code and does not build your application. Continuous Integration (CI) workflow is an area where different users and projects have their own requirements and preferences, so it supports layering CI workflows on Kubernetes but doesn't dictate how layering should work.

- Allows users to choose their logging, monitoring, and alerting systems. (It provides some integrations as proof of concept.)
- Does not provide nor mandate a comprehensive application configuration language/system (for example, jsonnet).
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.

On the other hand, a number of PaaS systems run *on* Kubernetes, such as Openshift, Deis, and Eldarion. You can also roll your own custom PaaS, integrate with a CI system of your choice, or use only Kubernetes by deploying your container images on Kubernetes.

Since Kubernetes operates at the application level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, logging, and monitoring. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable.

Additionally, Kubernetes is not a mere *orchestration system*. In fact, it eliminates the need for orchestration. The technical definition of *orchestration* is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes is comprised of a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required; the approach is more akin to *choreography*. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

What does *Kubernetes* mean? K8s?

The name **Kubernetes** originates from Greek, meaning *helmsman* or *pilot*, and is the root of *governor* and cybernetic. *K8s* is an abbreviation derived by replacing the 8 letters "ubernete" with "8".

What's next

- Ready to Get Started?
- For more details, see the Kubernetes Documentation.

Kubernetes Components

This document outlines the various binary components needed to deliver a functioning Kubernetes cluster.

Master Components

Master components provide the cluster's control plane. Master components make global decisions about the cluster (for example, scheduling), and detecting and responding to cluster events (starting up a new pod when a replication controller's 'replicas' field is unsatisfied).

Master components can be run on any node in the cluster. However, for simplicity, set up scripts typically start all master components on the same VM, and do not run user containers on this VM. See Building High-Availability Clusters for an example multi-master-VM setup.

kube-apiserver

kube-apiserver exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally – that is, it scales by deploying more instances. See Building High-Availability Clusters.

etcd

etcd is used as Kubernetes' backing store. All cluster data is stored here. Always have a backup plan for etcd's data for your Kubernetes cluster.

kube-controller-manager

kube-controller-manager runs controllers, which are the background threads that handle routine tasks in the cluster. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

cloud-controller-manager

cloud-controller-manager runs controllers that interact with the underlying cloud providers. The cloud-controller-manager binary is an alpha feature introduced in Kubernetes release 1.6.

cloud-controller-manager runs cloud-provider-specific controller loops only. You must disable these controller loops in the kube-controller-manager. You can disable the controller loops by setting the `--cloud-provider` flag to `external` when starting the kube-controller-manager.

cloud-controller-manager allows cloud vendors code and the Kubernetes core to evolve independent of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to cloud-controller-manager while running Kubernetes.

The following controllers have cloud provider dependencies:

- Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route Controller: For setting up routes in the underlying cloud infrastructure
- Service Controller: For creating, updating and deleting cloud provider load balancers
- Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

kube-scheduler

kube-scheduler watches newly created pods that have no node assigned, and selects a node for them to run on.

addons

Addons are pods and services that implement cluster features. The pods may be managed by Deployments, ReplicationControllers, and so on. Namespaced addon objects are created in the `kube-system` namespace.

Addon manager creates and maintains addon resources. See here for more details.

DNS

While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

Cluster-level Logging

A Cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

kubelet is the primary node agent. It watches for pods that have been assigned to its node (either by apiserver or via local configuration file) and:

- Mounts the pod's required volumes.
- Downloads the pod's secrets.
- Runs the pod's containers via docker (or, experimentally, rkt).
- Periodically executes any requested container liveness probes.
- Reports the status of the pod back to the rest of the system, by creating a *mirror pod* if necessary.
- Reports the status of the node back to the rest of the system.

kube-proxy

kube-proxy enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.

docker

`docker` is used for running containers.

rkt

`rkt` is supported experimentally for running containers as an alternative to docker.

supervisord

`supervisord` is a lightweight process monitor and control system that can be used to keep kubelet and docker running.

fluentd

`fluentd` is a daemon which helps provide cluster-level logging.

Understanding Kubernetes Objects

This page explains how Kubernetes objects are represented in the Kubernetes API, and how you can express them in `.yaml` format.

Understanding Kubernetes Objects

Kubernetes Objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's **desired state**.

To work with Kubernetes objects–whether to create, modify, or delete them–you'll need to use the Kubernetes API. When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you; you can also use the Kubernetes API directly in your own programs. Kubernetes currently provides a `golang` client library for this purpose, and other language libraries (such as Python) are being developed.

Object Spec and Status

Every Kubernetes object includes two nested object fields that govern the object’s configuration: the object *spec* and the object *status*. The *spec*, which you must provide, describes your *desired state* for the object–the characteristics that you want the object to have. The *status* describes the *actual state* of the object, and is supplied and updated by the Kubernetes system. At any given time, the Kubernetes Control Plane actively manages an object’s actual state to match the desired state you supplied.

For example, a Kubernetes Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment spec to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application–updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction–in this case, starting a replacement instance.

For more information on the object spec, status, and metadata, see the Kubernetes API Conventions.

Describing a Kubernetes Object

When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via `kubectl`), that API request must include that information as JSON in the request body. **Most often, you provide the information to `kubectl` in a .yaml file.** `kubectl` converts the information to JSON when making the API request.

Here’s an example `.yaml` file that shows the required fields and object spec for a Kubernetes Deployment:

nginx-deployment.yaml

nginx-deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

One way to create a Deployment using a `.yaml` file like the one above is to use the `kubectl create` command in the `kubectl` command-line interface, passing the `.yaml` file as an argument. Here's an example:

```
$ kubectl create -f docs/user-guide/nginx-deployment.yaml --record
```

The output is similar to this:

```
deployment "nginx-deployment" created
```

Required Fields

In the `.yaml` file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- `apiVersion` - Which version of the Kubernetes API you're using to create this object
- `kind` - What kind of object you want to create

- `metadata` - Data that helps uniquely identify the object, including a `name` string, UID, and optional `namespace`

You'll also need to provide the object `spec` field. The precise format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object. The Kubernetes API reference can help you find the spec format for all of the objects you can create using Kubernetes.

What's next

- Learn about the most important basic Kubernetes objects, such as Pod.

Names

All objects in the Kubernetes REST API are unambiguously identified by a Name and a UID.

For non-unique user-provided attributes, Kubernetes provides labels and annotations.

Names

Names are generally client-provided. Only one object of a given kind can have a given name at a time (i.e., they are spatially unique). But if you delete an object, you can make a new object with the same name. Names are used to refer to an object in a resource URL, such as `/api/v1/pods/some-name`. By convention, the names of Kubernetes resources should be up to maximum length of 253 characters and consist of lower case alphanumeric characters, `-`, and `.`, but certain resources have more specific restrictions. See the identifiers design doc for the precise syntax rules for names.

UIDs

UIDs are generated by Kubernetes. Every object created over the whole lifetime of a Kubernetes cluster has a distinct UID (i.e., they are spatially and temporally unique).

Namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.

Namespaces are a way to divide cluster resources between multiple uses (via resource quota).

In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

Working with Namespaces

Creation and deletion of namespaces is described in the Admin Guide documentation for namespaces.

Viewing namespaces

You can list the current namespaces in a cluster using:

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
```

Kubernetes starts with two initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system

Setting the namespace for a request

To temporarily set the namespace for a request, use the `--namespace` flag.

For example:

```
$ kubectl --namespace=<insert-namespace-name-here> run nginx --image=nginx
$ kubectl --namespace=<insert-namespace-name-here> get pods
```

Setting the namespace preference

You can permanently save the namespace for all subsequent kubectl commands in that context.

```
$ kubectl config set-context $(kubectl config current-context) --namespace=<insert
# Validate it
$ kubectl config view | grep namespace:
```

Namespaces and DNS

When you create a Service, it creates a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container just uses `<service-name>`, it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and

Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

Not All Objects are in a Namespace

Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace. Events are an exception: they may or may not have a namespace, depending on the object the event is about.

Labels and Selectors

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

We'll eventually index and reverse-index labels for efficient queries and watches, use them to sort and group in UIs and CLIs, etc. We don't want to pollute labels with non-identifying, especially large and/or structured, data. Non-identifying information should be recorded using annotations.

Motivation

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings.

Service deployments and batch processing pipelines are often multi-dimensional entities (e.g., multiple partitions or deployments, multiple release tracks, multiple tiers, multiple micro-services per tier). Management often requires cross-cutting operations, which breaks encapsulation of strictly hierarchical representations, especially rigid hierarchies determined by the infrastructure rather than by users.

Example labels:

- `"release" : "stable"`, `"release" : "canary"`
- `"environment" : "dev"`, `"environment" : "qa"`, `"environment" : "production"`
- `"tier" : "frontend"`, `"tier" : "backend"`, `"tier" : "cache"`
- `"partition" : "customerA"`, `"partition" : "customerB"`
- `"track" : "daily"`, `"track" : "weekly"`

These are just examples of commonly used labels; you are free to develop your own conventions. Keep in mind that label Key must be unique for a given object.

Syntax and character set

Labels are key/value pairs. Valid label keys have two segments: an optional prefix and name, separated by a slash (`/`). The name segment is required and must be 63 characters or less, beginning and ending with an alphanumeric character (`[a-z0-9A-Z]`) with dashes (`-`), underscores (`_`), dots (`.`), and alphanumerics between. The prefix is optional. If specified, the prefix must be a DNS subdomain: a series of DNS labels separated by dots (`.`), not longer than 253 characters in total, followed by a slash (`/`). If the prefix is omitted, the label Key is presumed to be private to the user. Automated system components (e.g. `kube-scheduler`, `kube-controller-manager`, `kube-apiserver`, `kubectl`, or other third-party automation) which add labels to end-user objects must specify a prefix. The `kubernetes.io/` prefix is reserved for Kubernetes core components.

Valid label values must be 63 characters or less and must be empty or begin and end with an alphanumeric character (`[a-z0-9A-Z]`) with dashes (`-`), underscores (`_`), dots (`.`), and alphanumerics between.

Label selectors

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

The API currently supports two types of selectors: *equality-based* and *set-based*. A label selector can be made of multiple *requirements* which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as an *AND* logical operator.

An empty label selector (that is, one with zero requirements) selects every object in the collection.

A null label selector (which is only possible for optional selector fields) selects no objects.

Note: the label selectors of two controllers must not overlap within a namespace, otherwise they will fight with each other.

Equality-based requirement

Equality- or *inequality-based* requirements allow filtering by label keys and values. Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well. Three kinds of operators are admitted `=`, `==`, `!=`. The first two represent *equality* (and are simply synonyms), while the latter represents *inequality*. For example:

```
environment = production
tier != frontend
```

The former selects all resources with key equal to `environment` and value equal to `production`. The latter selects all resources with key equal to `tier` and value distinct from `frontend`, and all resources with no labels with the `tier` key. One could filter for resources in `production` excluding `frontend` using the comma operator: `environment=production,tier!=frontend`

Set-based requirement

Set-based label requirements allow filtering keys according to a set of values. Three kinds of operators are supported: `in`, `notin` and `exists` (only the key identifier). For example:

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

The first example selects all resources with key equal to `environment` and value equal to `production` or `qa`. The second example selects all resources with key equal to `tier` and values other than `frontend` and `backend`, and all resources with no labels with the `tier` key. The third example selects all resources including a label with key `partition`; no values are checked. The fourth example selects all resources without a label with key `partition`; no values are checked. Similarly the comma separator acts as an *AND* operator. So filtering resources with a `partition` key (no matter the value) and with `environment` different than `qa` can be achieved using `partition,environment notin (qa)`. The *set-based* label selector is a general form of equality since `environment=production` is equivalent to `environment in (production)`; similarly for `!=` and `notin`.

Set-based requirements can be mixed with *equality-based* requirements. For example: `partition in (customerA, customerB),environment!=qa`.

API

LIST and WATCH filtering

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter. Both requirements are permitted (presented here as they would appear in a URL query string):

- *equality-based* requirements: `?labelSelector=environment%3Dproduction,tier%3Dfrontend`
- *set-based* requirements: `?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29`

Both label selector styles can be used to list or watch resources via a REST client. For example, targeting `apiserver` with `kubectl` and using *equality-based* one may write:

```
$ kubectl get pods -l environment=production,tier=frontend
```

or using *set-based* requirements:

```
$ kubectl get pods -l 'environment in (production),tier in (frontend)'
```

As already mentioned *set-based* requirements are more expressive. For instance, they can implement the *OR* operator on values:

```
$ kubectl get pods -l 'environment in (production, qa)'
```

or restricting negative matching via *exists* operator:

```
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

Set references in API objects

Some Kubernetes objects, such as `services` and `replicationcontrollers`, also use label selectors to specify sets of other resources, such as pods.

Service and ReplicationController

The set of pods that a `service` targets is defined with a label selector. Similarly, the population of pods that a `replicationcontroller` should manage is also defined with a label selector.

Labels selectors for both objects are defined in `json` or `yaml` files using maps, and only *equality-based* requirement selectors are supported:

```
"selector": {
    "component" : "redis",
}
```

or

```
selector:
    component: redis
```

this selector (respectively in `json` or `yaml` format) is equivalent to `component=redis` or `component in (redis)`.

Resources that support set-based requirements

Newer resources, such as `Job`, `Deployment`, `Replica Set`, and `Daemon Set`, support *set-based* requirements as well.

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

`matchLabels` is a map of `{key,value}` pairs. A single `{key,value}` in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose `key` field is "key", the `operator` is "In", and the `values` array contains only "value". `matchExpressions` is a list of pod selector requirements. Valid operators include In, NotIn, Exists, and DoesNotExist. The values set must be non-empty in the case of In and NotIn. All of the requirements, from both `matchLabels` and `matchExpressions` are ANDed together – they must all be satisfied in order to match.

Selecting sets of nodes

One use case for selecting over labels is to constrain the set of nodes onto which a pod can schedule. See the documentation on node selection for more information.

Annotations

You can use Kubernetes annotations to attach arbitrary non-identifying metadata to objects. Clients such as tools and libraries can retrieve this metadata.

Attaching metadata to objects

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. The metadata in an annotation can be small or large, structured or unstructured, and can include characters not permitted by labels.

Annotations, like labels, are key/value maps:

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Here are some examples of information that could be recorded in annotations:

- Fields managed by a declarative configuration layer. Attaching these fields as annotations distinguishes them from default values set by clients or servers, and from auto-generated fields and fields set by auto-sizing or auto-scaling systems.
- Build, release, or image information like timestamps, release IDs, git branch, PR numbers, image hashes, and registry address.
- Pointers to logging, monitoring, analytics, or audit repositories.
- Client library or tool information that can be used for debugging purposes: for example, name, version, and build information.

- User or tool/system provenance information, such as URLs of related objects from other ecosystem components.
- Lightweight rollout tool metadata: for example, config or checkpoints.
- Phone or pager numbers of persons responsible, or directory entries that specify where that information can be found, such as a team web site.

Instead of using annotations, you could store this type of information in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, and the like.

What’s next

Learn more about Labels and Selectors.

The Kubernetes API

Overall API conventions are described in the API conventions doc.

API endpoints, resource types and samples are described in API Reference.

Remote access to the API is discussed in the access doc.

The Kubernetes API also serves as the foundation for the declarative configuration schema for the system. The Kubectl command-line tool can be used to create, update, delete, and get API objects.

Kubernetes also stores its serialized state (currently in etcd) in terms of the API resources.

Kubernetes itself is decomposed into multiple components, which interact through its API.

API changes

In our experience, any system that is successful needs to grow and change as new use cases emerge or existing ones change. Therefore, we expect the Kubernetes API to continuously change and grow. However, we intend to not break compatibility with existing clients, for an extended period of time. In general, new API resources and new resource fields can be expected to be added frequently. Elimination of resources or fields will require following the API deprecation policy.

What constitutes a compatible change and how to change the API are detailed by the API change document.

OpenAPI and Swagger definitions

Complete API details are documented using Swagger v1.2 and OpenAPI. The Kubernetes apiserver (aka "master") exposes an API that can be used to retrieve the Swagger v1.2 Kubernetes API spec located at `/swaggerapi`. You can also enable a UI to browse the API documentation at `/swagger-ui` by passing the `--enable-swagger-ui=true` flag to apiserver.

Starting with Kubernetes 1.4, OpenAPI spec is also available at `/swagger.json`. While we are transitioning from Swagger v1.2 to OpenAPI (aka Swagger v2.0), some of the tools such as kubectl and swagger-ui are still using v1.2 spec. OpenAPI spec is in Beta as of Kubernetes 1.5.

Kubernetes implements an alternative Protobuf based serialization format for the API that is primarily intended for intra-cluster communication, documented in the design proposal and the IDL files for each schema are located in the Go packages that define the API objects.

API versioning

To make it easier to eliminate fields or restructure resource representations, Kubernetes supports multiple API versions, each at a different API path, such as `/api/v1` or `/apis/extensions/v1beta1`.

We chose to version at the API level rather than at the resource or field level to ensure that the API presents a clear, consistent view of system resources and behavior, and to enable controlling access to end-of-lifed and/or experimental APIs. The JSON and Protobuf serialization schemas follow the same guidelines for schema changes - all descriptions below cover both formats.

Note that API versioning and Software versioning are only indirectly related. The API and release versioning proposal describes the relationship between API versioning and software versioning.

Different API versions imply different levels of stability and support. The criteria for each level are described in more detail in the API Changes documentation. They are summarized here:

- Alpha level:
  - The version names contain `alpha` (e.g. `v1alpha1`).
  - May be buggy. Enabling the feature may expose bugs. Disabled by default.
  - Support for feature may be dropped at any time without notice.
  - The API may change in incompatible ways in a later software release without notice.
  - Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.
- Beta level:
  - The version names contain `beta` (e.g. `v2beta3`).
  - Code is well tested. Enabling the feature is considered safe. Enabled by default.

  - Support for the overall feature will not be dropped, though details may change.
  - The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
  - Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters which can be upgraded independently, you may be able to relax this restriction.
  - **Please do try our beta features and give feedback on them! Once they exit beta, it may not be practical for us to make more changes.**
- Stable level:
  - The version name is `vX` where `X` is an integer.
  - Stable versions of features will appear in released software for many subsequent versions.

API groups

To make it easier to extend the Kubernetes API, we implemented *API groups*. The API group is specified in a REST path and in the `apiVersion` field of a serialized object.

Currently there are several API groups in use:

1. The "core" (oftentimes called "legacy", due to not having explicit group name) group, which is at REST path `/api/v1` and is not specified as part of the `apiVersion` field, e.g. `apiVersion: v1` .
2. The named groups are at REST path `/apis/$GROUP_NAME/$VERSION`, and use `apiVersion: $GROUP_NAME/$VERSION` (e.g. `apiVersion: batch/v1`). Full list of supported API groups can be seen in Kubernetes API reference.

There are two supported paths to extending the API with custom resources:

1. CustomResourceDefinition is for users with very basic CRUD needs.

2. Coming soon: users needing the full set of Kubernetes API semantics can implement their own apiserver and use the aggregator to make it seamless for clients.

Enabling API groups

Certain resources and API groups are enabled by default. They can be enabled or disabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For ex: to disable batch/v1, set `--runtime-config=batch/v1=false`, to enable batch/v2alpha1, set `--runtime-config=batch/v2alpha1`. The flag accepts comma separated set of key=value pairs describing runtime configuration of the apiserver.

IMPORTANT: Enabling or disabling groups or resources requires restarting apiserver and controller-manager to pick up the `--runtime-config` changes.

Enabling resources in the groups

DaemonSets, Deployments, HorizontalPodAutoscalers, Ingress, Jobs and ReplicaSets are enabled by default. Other extensions resources can be enabled by setting `--runtime-config` on apiserver. `--runtime-config` accepts comma separated values. For example: to disable deployments and ingress, set

```
--runtime-config=extensions/v1beta1/deployments=false,extensions/v1beta1/ingress=false
```

Nodes

What is a node?

A `node` is a worker machine in Kubernetes, previously known as a `minion`. A node may be a VM or physical machine, depending on the cluster. Each node has the services necessary to run pods and is managed by the master components. The services on a node include Docker, kubelet and kube-proxy. See The Kubernetes Node section in the architecture design doc for more details.

Node Status

A node's status contains the following information:

- Addresses
- ~~Phase~~ **deprecated**
- Condition
- Capacity

- Info

Each section is described in detail below.

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet `--hostname-override` parameter.
- ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
- InternalIP: Typically the IP address of the node that is routable only within the cluster.

Phase

Deprecated: node phase is no longer used.

Condition

The `conditions` field describes the status of all `Running` nodes.

Node Condition	Description
`OutOfDisk`	`True` if there is insufficient free space on the node for adding new pods, otherwise `False`
`Ready`	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy and is not accepting pods, and `Unknown` if the node controller has not heard from the node in the last 40 seconds
`MemoryPressure`	`True` if pressure exists on the node memory – that is, if the node memory is low; otherwise `False`
`DiskPressure`	`True` if pressure exists on the disk size – that is, if the disk capacity is low; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`

The node condition is represented as a JSON object. For example, the following response describes a healthy node.

```
"conditions": [
  {
    "kind": "Ready",
    "status": "True"
  }
]
```

If the Status of the Ready condition is “Unknown” or “False” for longer than the `pod-eviction-timeout`, an argument is passed to the kube-controller-manager and all of the Pods on the node are scheduled for deletion by the Node Controller. The default eviction timeout duration is **five minutes**. In some cases when the node is unreachable, the apiserver is unable to communicate with the kubelet on it. The decision to delete the pods cannot be communicated to the kubelet until it re-establishes communication with the apiserver. In the meantime, the pods which are scheduled for deletion may continue to run on the partitioned node.

In versions of Kubernetes prior to 1.5, the node controller would force delete these unreachable pods from the apiserver. However, in 1.5 and higher, the node controller does not force delete pods until it is confirmed that they have stopped running in the cluster. One can see these pods which may be running on an unreachable node as being in the “Terminating” or “Unknown” states. In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand. Deleting the node object from Kubernetes causes all the Pod objects running on it to be deleted from the apiserver, freeing up their names.

Version 1.8 introduces an alpha feature that automatically creates taints that represent conditions. To enable this behavior, pass an additional feature gate flag `--feature-gates=...,TaintNodesByCondition=true` to the API server, controller manager, and scheduler. When `TaintNodesByCondition` is enabled, the scheduler ignores conditions when considering a Node; instead it looks at the Node’s taints and a Pod’s tolerations.

Now users can choose between the old scheduling model and a new, more flexible scheduling model. A Pod that does not have any tolerations gets scheduled according to the old model. But a Pod that tolerates the taints of a particular Node can be scheduled on that Node.

Note that because of small delay, usually less than one second, between time when condition is observed and a taint is created, it's possible that enabling this feature will slightly increase number of Pods that are successfully scheduled but rejected by the kubelet.

Capacity

Describes the resources available on the node: CPU, memory and the maximum number of pods that can be scheduled onto the node.

Info

General information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), OS name. The information is gathered by Kubelet from the node.

Management

Unlike pods and services, a node is not inherently created by Kubernetes: it is created externally by cloud providers like Google Compute Engine, or exists in your pool of physical or virtual machines. What this means is that when Kubernetes creates a node, it is really just creating an object that represents the node. After creation, Kubernetes will check whether the node is valid or not. For example, if you try to create a node from the following content:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes will create a node object internally (the representation), and validate the node by health checking based on the `metadata.name` field (we assume `metadata.name` can be resolved). If the node is valid, i.e. all necessary services are running, it is eligible to run a pod; otherwise, it will be ignored for any cluster activity until it becomes valid. Note that Kubernetes will keep the object for the invalid node unless it is explicitly deleted by the client, and it will keep checking to see if it becomes valid.

Currently, there are three components that interact with the Kubernetes node interface: node controller, kubelet, and kubectl.

Node Controller

The node controller is a Kubernetes master component which manages various aspects of nodes.

The node controller has multiple roles in a node's life. The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment, whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

The third is monitoring the nodes' health. The node controller is responsible for updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable (i.e. the node controller stops receiving heartbeats for some reason, e.g. due to the node being down), and then later evicting all the pods from the node (using graceful termination) if the node continues to be unreachable. (The default timeouts are 40s to start reporting ConditionUnknown and 5m after that to start evicting pods.) The node controller checks the state of each node every `--node-monitor-period` seconds.

In Kubernetes 1.4, we updated the logic of the node controller to better handle cases when a large number of nodes have problems with reaching the master (e.g. because the master has networking problem). Starting with 1.4, the node controller will look at the state of all nodes in the cluster when making a decision about pod eviction.

In most cases, node controller limits the eviction rate to `--node-eviction-rate` (default 0.1) per second, meaning it won't evict pods from more than 1 node per 10 seconds.

The node eviction behavior changes when a node in a given availability zone becomes unhealthy. The node controller checks what percentage of nodes in the zone are unhealthy (NodeReady condition is ConditionUnknown or ConditionFalse) at the same time. If the fraction of unhealthy nodes is at least `--unhealthy-zone-threshold` (default 0.55) then the eviction rate is reduced: if the cluster is small (i.e. has less than or equal to `--large-cluster-size-threshold` nodes - default 50) then evictions are stopped, otherwise the eviction rate is reduced to `--secondary-node-eviction-rate` (default 0.01) per second. The reason these policies are implemented per availability zone is because one availability zone might become partitioned from the master while the others remain connected. If your cluster does not span multiple cloud provider availability zones, then there is only one availability zone (the whole cluster).

A key reason for spreading your nodes across availability zones is so that the workload can be shifted to healthy zones when one entire zone goes down. Therefore, if all nodes in a zone are unhealthy then node controller evicts at the normal rate `--node-eviction-rate`. The corner case is when all zones are completely unhealthy (i.e. there are no healthy nodes in the cluster). In such case, the node controller assumes that there's some problem with master connectivity and stops all evictions until some connectivity is restored.

Starting in Kubernetes 1.6, the NodeController is also responsible for evicting pods that are running on nodes with `NoExecute` taints, when the pods do not tolerate the taints. Additionally, as an alpha feature that is disabled by default, the NodeController is responsible for adding taints corresponding to node problems like node unreachable or not ready. See this documentation for details about `NoExecute` taints and the alpha feature.

Starting in version 1.8, the node controller can be made responsible for creating taints that represent Node conditions. This is an alpha feature of version 1.8.

Self-Registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the apiserver.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`). No-op if `register-node` is false.

- `--node-ip` IP address of the node.
- `--node-labels` - Labels to add when registering the node in the cluster.
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

Currently, any kubelet is authorized to create/modify any node resource, but in practice it only creates/modifies its own. (In the future, we plan to only allow a kubelet to modify its own node resource.)

Manual Node Administration

A cluster administrator can create and modify node objects.

If the administrator wishes to create node objects manually, set the kubelet flag `--register-node=false`.

The administrator can modify node resources (regardless of the setting of `--register-node`). Modifications include setting labels on the node and marking it unschedulable.

Labels on nodes can be used in conjunction with node selectors on pods to control scheduling, e.g. to constrain a pod to only be eligible to run on a subset of the nodes.

Marking a node as unschedulable will prevent new pods from being scheduled to that node, but will not affect any existing pods on the node. This is useful as a preparatory step before a node reboot, etc. For example, to mark a node unschedulable, run this command:

```
kubectl cordon $NODENAME
```

Note that pods which are created by a DaemonSet controller bypass the Kubernetes scheduler, and do not respect the unschedulable attribute on a node. The assumption is that daemons belong on the machine even if it is being drained of applications in preparation for a reboot.

Node capacity

The capacity of the node (number of cpus and amount of memory) is part of the node object. Normally, nodes register themselves and report their capacity when creating the node object. If you are doing manual node administration, then you need to set node capacity when adding a node.

The Kubernetes scheduler ensures that there are enough resources for all the pods on a node. It checks that the sum of the requests of containers on the node is no greater than the node capacity. It includes all containers started by the kubelet, but not containers started directly by Docker nor processes not in containers.

If you want to explicitly reserve resources for non-pod processes, you can create a placeholder pod. Use the following template:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: sleep-forever
    image: gcr.io/google_containers/pause:0.8.0
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
```

Set the `cpu` and `memory` values to the amount of resources you want to reserve. Place the file in the manifest directory (`--config=DIR` flag of kubelet). Do this on each kubelet where you want to reserve resources.

API Object

Node is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Node API object.

Master-Node communication

Overview

This document catalogs the communication paths between the master (really the apiserver) and the Kubernetes cluster. The intent is to allow users to customize their installation to harden the network configuration such that the cluster can be run on an untrusted network (or on fully public IPs on a cloud provider).

Cluster -> Master

All communication paths from the cluster to the master terminate at the apiserver (none of the other master components are designed to expose remote services). In a typical deployment, the apiserver is configured to listen for remote connections on a secure HTTPS port (443) with one or more forms of client authentication enabled. One or more forms of authorization should be enabled, especially if anonymous requests or service account tokens are allowed.

Nodes should be provisioned with the public root certificate for the cluster such that they can connect securely to the apiserver along with valid client credentials. For example, on a default GCE deployment, the client credentials provided to the kubelet are in the form of a client certificate. See kubelet TLS bootstrapping for automated provisioning of kubelet client certificates.

Pods that wish to connect to the apiserver can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The `kubernetes` service (in all namespaces) is configured with a virtual IP address that is redirected (via kube-proxy) to the HTTPS endpoint on the apiserver.

The master components communicate with the cluster apiserver over the insecure (not encrypted or authenticated) port. This port is typically only exposed on the localhost interface of the master machine, so that the master components, all running on the same machine, can communicate with the cluster apiserver. Over time, the master components will be migrated to use the secure port with authentication and authorization (see #13598).

As a result, the default operating mode for connections from the cluster (nodes and pods running on the nodes) to the master is secured by default and can run over untrusted and/or public networks.

Master -> Cluster

There are two primary communication paths from the master (apiserver) to the cluster. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality.

apiserver -> kubelet

The connections from the apiserver to the kubelet are used for:

- Fetching logs for pods.
- Attaching (through kubectl) to running pods.
- Providing the kubelet's port-forwarding functionality.

These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks, and **unsafe** to run over untrusted and/or public networks.

To verify this connection, use the `--kubelet-certificate-authority` flag to provide the apiserver with a root certificate bundle to use to verify the kubelet's serving certificate.

If that is not possible, use SSH tunneling between the apiserver and kubelet if required to avoid connecting over an untrusted or public network.

Finally, Kubelet authentication and/or authorization should be enabled to secure the kubelet API.

apiserver -> nodes, pods, and services

The connections from the apiserver to a node, pod, or service default to plain HTTP connections and are therefore neither authenticated nor encrypted. They can be run over a secure HTTPS connection by prefixing `https:` to the node, pod, or service name in the API URL, but they will not validate the certificate provided by the HTTPS endpoint nor provide client credentials so while the connection will be encrypted, it will not provide any guarantees of integrity. These connections **are not currently safe** to run over untrusted and/or public networks.

SSH Tunnels

Google Container Engine uses SSH tunnels to protect the Master -> Cluster communication paths. In this configuration, the apiserver initiates an SSH tunnel to each node in the cluster (connecting to the ssh server listening on port 22) and passes all traffic destined for a kubelet, node, pod, or service through the tunnel. This tunnel ensures that the traffic is not exposed outside of the private GCE network in which the cluster is running.

Custom Resources

This page explains the concept of *custom resources*, which are extensions of the Kubernetes API.

Custom resources

A *resource* is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind. For example, the built-in *pods* resource contains a collection of Pod objects.

A *custom resource* is an extension of the Kubernetes API that is not necessarily available on every Kubernetes cluster. In other words, it represents a customization of a particular Kubernetes installation.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects with kubectl, just as they do for built-in resources like *pods*.

Custom controllers

On their own, custom resources simply let you store and retrieve structured data. It is only when combined with a *controller* that they become a true declarative API. The controller interprets the structured data as a record of the user's desired state, and continually takes action to achieve and maintain that state.

A *custom controller* is a controller that users can deploy and update on a running cluster, independently of the cluster's own lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The Operator pattern is one example of such a combination. It allows developers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

CustomResourceDefinitions

CustomResourceDefinition (CRD) is a built-in API that offers a simple way to create custom resources. Deploying a CRD into the cluster causes the Kubernetes API server to begin serving the specified custom resource on your behalf.

This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with API server aggregation.

Refer to the Custom Resource Example for a demonstration of how to register a new custom resource, work with instances of your new resource type, and setup a controller to handle events.

> **Note:** CRD is the successor to the deprecated *ThirdPartyResource* (TPR) API, and is available as of Kubernetes 1.7.

API server aggregation

Usually, each resource in the Kubernetes API requires code that handles REST requests and manages persistent storage of objects. The main Kubernetes API server handles built-in resources like *pods* and *services*, and can also handle custom resources in a generic way through CustomResourceDefinitions.

The aggregation layer allows you to provide specialized implementations for your custom resources by writing and deploying your own standalone API server. The main API server delegates requests to you for the custom resources that you handle, making them available to all of its clients.

What's next

- Learn how to Extend the Kubernetes API with the aggregation layer.

- Learn how to Extend the Kubernetes API with CustomResourceDefinition.
- Learn how to Migrate a ThirdPartyResource to CustomResourceDefinition.

Extending the Kubernetes API with the aggregation layer

The aggregation layer allows Kubernetes to be extended with additional APIs, beyond what is offered by the core Kubernetes APIs.

Overview

The aggregation layer enables installing additional Kubernetes-style APIs in your cluster. These can either be pre-built, existing 3rd party solutions, such as service-catalog, or user-created APIs like apiserver-builder, which can get you started.

In 1.7 the aggregation layer runs in-process with the kube-apiserver. Until an extension resource is registered, the aggregation layer will do nothing. To register an API, users must add an APIService object, which "claims" the URL path in the Kubernetes API. At that point, the aggregation layer will proxy anything sent to that API path (e.g. /apis/myextension.mycompany.io/v1/…) to the registered APIService.

Ordinarily, the APIService will be implemented by an *extension-apiserver* in a pod running in the cluster. This extension-apiserver will normally need to be paired with one or more controllers if active management of the added resources is needed. As a result, the apiserver-builder will actually provide a skeleton for both. As another example, when the service-catalog is installed, it provides both the extension-apiserver and controller for the services it provides.

What's next

- To get the aggregator working in your environment, configure the aggregation layer.
- Then, setup an extension api-server to work with the aggregation layer.

- Also, learn how to extend the Kubernetes API using Custom Resource Definitions.

Images

You create your Docker image and push it to a registry before referring to it in a Kubernetes pod.

The `image` property of a container supports the same syntax as the `docker` command does, including private registries and tags.

Updating Images

The default pull policy is `IfNotPresent` which causes the Kubelet to skip pulling an image if it already exists. If you would like to always force a pull, you can do one of the following:

- set the `imagePullPolicy` of the container to `Always`;
- use `:latest` as the tag for the image to use;
- enable the AlwaysPullImages admission controller.

If you did not specify tag of your image, it will be assumed as `:latest`, with pull image policy of `Always` correspondingly.

Note that you should avoid using `:latest` tag, see Best Practices for Configuration for more information.

Using a Private Registry

Private registries may require keys to read images from them. Credentials can be provided in several ways:

- Using Google Container Registry
  - Per-cluster
  - automatically configured on Google Compute Engine or Google Container Engine
  - all pods can read the project's private registry
- Using AWS EC2 Container Registry (ECR)
  - use IAM roles and policies to control access to ECR repositories
  - automatically refreshes ECR login credentials
- Using Azure Container Registry (ACR)
- Configuring Nodes to Authenticate to a Private Registry
  - all pods can read any configured private registries
  - requires node configuration by cluster administrator
- Pre-pulling Images
  - all pods can use any images cached on a node
  - requires root access to all nodes to setup
- Specifying ImagePullSecrets on a Pod
  - only pods which provide own keys can access the private registry Each option is described in more detail below.

Using Google Container Registry

Kubernetes has native support for the [Google Container Registry (GCR)](), when running on Google Compute Engine (GCE). If you are running your cluster on GCE or Google Container Engine (GKE), simply use the full image name (e.g. gcr.io/my_project/image:tag).

All pods in a cluster will have read access to images in this registry.

The kubelet will authenticate to GCR using the instance's Google service account. The service account on the instance will have a `https://www.googleapis.com/auth/devstorage.read_only`, so it can pull from the project's GCR, but not push.

Using AWS EC2 Container Registry

Kubernetes has native support for the [AWS EC2 Container Registry](), when nodes are AWS EC2 instances.

Simply use the full image name (e.g. `ACCOUNT.dkr.ecr.REGION.amazonaws.com/imagename:tag`) in the Pod definition.

All users of the cluster who can create pods will be able to run pods that use any of the images in the ECR registry.

The kubelet will fetch and periodically refresh ECR credentials. It needs the following permissions to do this:

- `ecr:GetAuthorizationToken`
- `ecr:BatchCheckLayerAvailability`
- `ecr:GetDownloadUrlForLayer`
- `ecr:GetRepositoryPolicy`
- `ecr:DescribeRepositories`
- `ecr:ListImages`
- `ecr:BatchGetImage`

Requirements:

- You must be using kubelet version `v1.2.0` or newer. (e.g. run `/usr/bin/kubelet --version=true`).
- If your nodes are in region A and your registry is in a different region B, you need version `v1.3.0` or newer.
- ECR must be offered in your region

Troubleshooting:

- Verify all requirements above.
- Get $REGION (e.g. `us-west-2`) credentials on your workstation. SSH into the host and run Docker manually with those creds. Does it work?
- Verify kubelet is running with `--cloud-provider=aws`.
- Check kubelet logs (e.g. `journalctl -u kubelet`) for log lines like:
  - `plugins.go:56] Registering credential provider: aws-ecr-key`
  - `provider.go:91] Refreshing cache for provider: *aws_credentials.ecrProvider`

Using Azure Container Registry (ACR)

When using Azure Container Registry you can authenticate using either an admin user or a service principal. In either case, authentication is done via standard Docker authentication. These instructions assume the azure-cli command line tool.

You first need to create a registry and generate credentials, complete documentation for this can be found in the Azure container registry documentation.

Once you have created your container registry, you will use the following credentials to login:

- `DOCKER_USER` : service principal, or admin username
- `DOCKER_PASSWORD`: service principal password, or admin user password
- `DOCKER_REGISTRY_SERVER`: `${some-registry-name}.azurecr.io`
- `DOCKER_EMAIL`: `${some-email-address}`

Once you have those variables filled in you can configure a Kubernetes Secret and use it to deploy a Pod.

Configuring Nodes to Authenticate to a Private Repository

Note: if you are running on Google Container Engine (GKE), there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.

Note: if you are running on AWS EC2 and are using the EC2 Container Registry (ECR), the kubelet on each node will manage and update the ECR login credentials. You cannot use this approach.

Note: this approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.

Docker stores keys for private registries in the `$HOME/.dockercfg` or `$HOME/.docker/config.json` file. If you put this in the `$HOME` of user `root` on a kubelet, then docker will use it.

Here are the recommended steps to configuring your nodes to use a private registry. In this example, run these on your desktop/laptop:

1. Run `docker login [server]` for each set of credentials you want to use. This updates `$HOME/.docker/config.json`.
2. View `$HOME/.docker/config.json` in an editor to ensure it contains just the credentials you want to use.
3. Get a list of your nodes, for example:
    1. if you want the names:
       ```
       nodes=$(kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}')
       ```
    2. if you want to get the IPs:
       ```
       nodes=$(kubectl get nodes -o jsonpath='{range .items[*].status.addresses[?(@.type=="ExternalIP")]}{.address} {end}')
       ```
4. Copy your local `.docker/config.json` to the home directory of root on each node.

1. for example:

```
for n in $nodes; do scp ~/.docker/config.json
root@$n:/root/.docker/config.json; done
```

Verify by creating a pod that uses a private image, e.g.:

```
$ cat <<EOF > /tmp/private-image-test-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: private-image-test-1
spec:
  containers:
    - name: uses-private-image
      image: $PRIVATE_IMAGE_NAME
      imagePullPolicy: Always
      command: [ "echo", "SUCCESS" ]
EOF
$ kubectl create -f /tmp/private-image-test-1.yaml
pod "private-image-test-1" created
$
```

If everything is working, then, after a few moments, you should see:

```
$ kubectl logs private-image-test-1
SUCCESS
```

If it failed, then you will see:

```
$ kubectl describe pods/private-image-test-1 | grep "Failed"
  Fri, 26 Jun 2015 15:36:13 -0700    Fri, 26 Jun 2015 15:39:13 -0700    19    {kub
```

You must ensure all nodes in the cluster have the same `.docker/config.json`. Otherwise, pods will run on some nodes and fail to run on others. For example, if you use node autoscaling, then each instance template needs to include the `.docker/config.json` or mount a drive that contains it.

All pods will have read access to images in any private registry once private registry keys are added to the `.docker/config.json`.

This was tested with a private docker repository as of 26 June with Kubernetes version v0.19.3. It should also work for a private registry such as quay.io, but that has not been tested.

Pre-pulling Images

Note: if you are running on Google Container Engine (GKE), there will already be a `.dockercfg` on each node with credentials for Google Container Registry. You cannot use this approach.

Note: this approach is suitable if you can control node configuration. It will not work reliably on GCE, and any other cloud provider that does automatic node replacement.

By default, the kubelet will try to pull each image from the specified registry. However, if the `imagePullPolicy` property of the container is set to `IfNotPresent` or `Never`, then a local image is used (preferentially or exclusively, respectively).

If you want to rely on pre-pulled images as a substitute for registry authentication, you must ensure all nodes in the cluster have the same pre-pulled images.

This can be used to preload certain images for speed or as an alternative to authenticating to a private registry.

All pods will have read access to any pre-pulled images.

Specifying ImagePullSecrets on a Pod

Note: This approach is currently the recommended approach for GKE, GCE, and any cloud-providers where node creation is automated.

Kubernetes supports specifying registry keys on a pod.

Creating a Secret with a Docker Config

Run the following command, substituting the appropriate uppercase values:

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGIS
secret "myregistrykey" created.
```

If you need access to multiple registries, you can create one secret for each registry. Kubelet will merge any `imagePullSecrets` into a single virtual `.docker/config.json` when pulling images for your Pods.

Pods can only reference image pull secrets in their own namespace, so this process needs to be done one time per namespace.

Bypassing kubectl create secrets

If for some reason you need multiple items in a single `.docker/config.json` or need control not given by the above command, then you can create a secret using json or yaml.

Be sure to:

- set the name of the data item to `.dockerconfigjson`
- base64 encode the docker file and paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
- set `type` to `kubernetes.io/dockerconfigjson`

Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
  namespace: awesomeapps
data:
  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWFhYWFhYWFhYWFhYWFhYWFhY
type: kubernetes.io/dockerconfigjson
```

If you get the error message `error: no objects passed to create`, it may mean the base64 encoded string is invalid. If you get an error message like `Secret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value ...`, it means the data was successfully un-base64 encoded, but could not be parsed as a `.docker/config.json` file.

Referring to an imagePullSecrets on a Pod

Now, you can create pods which reference that secret by adding an `imagePullSecrets` section to a pod definition.

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
  namespace: awesomeapps
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

This needs to be done for each pod that is using a private registry.

However, setting of this field can be automated by setting the imagePullSecrets in a serviceAccount resource.

You can use this in conjunction with a per-node `.docker/config.json`. The credentials will be merged. This approach will work on Google Container Engine (GKE).

Use Cases

There are a number of solutions for configuring private registries. Here are some common use cases and suggested solutions.

1. Cluster running only non-proprietary (e.g. open-source) images. No need to hide images.
    1. Use public images on the Docker hub.
        1. No configuration required.
        2. On GCE/GKE, a local mirror is automatically used for improved speed and availability.
2. Cluster running some proprietary images which should be hidden to those outside the company, but visible to all cluster users.
    1. Use a hosted private Docker registry.
        1. It may be hosted on the Docker Hub, or elsewhere.

        2. Manually configure .docker/config.json on each node as described above.
    2. Or, run an internal private registry behind your firewall with open read access.
        1. No Kubernetes configuration is required.
    3. Or, when on GCE/GKE, use the project's Google Container Registry.
        1. It will work better with cluster autoscaling than manual node configuration.
    4. Or, on a cluster where changing the node configuration is inconvenient, use `imagePullSecrets`.
3. Cluster with a proprietary images, a few of which require stricter access control.
    1. Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods potentially have access to all images.
    2. Move sensitive data into a "Secret" resource, instead of packaging it in an image.
4. A multi-tenant cluster where each tenant needs own private registry.
    1. Ensure AlwaysPullImages admission controller is active. Otherwise, all Pods of all tenants potentially have access to all images.
    2. Run a private registry with authorization required.
    3. Generate registry credential for each tenant, put into secret, and populate secret to each tenant namespace.
    4. The tenant adds that secret to imagePullSecrets of each namespace.

Container Environment Variables

This page describes the resources available to Containers in the Container environment.

Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

Container information

The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc.

The Pod name and namespace are available as environment variables through the downward API.

User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the Docker image.

Cluster information

A list of all services that were running when a Container was created is available to that Container as environment variables. Those environment variables match the syntax of Docker links.

For a service named *foo* that maps to a container port named *bar*, the following variables are defined:

```
FOO_SERVICE_HOST=<the host the service is running on>
FOO_SERVICE_PORT=<the port the service is running on>
```

Services have dedicated IP addresses and are available to the Container via DNS, if DNS addon is enabled.

What's next

- Learn more about Container lifecycle hooks.
- Get hands-on experience attaching handlers to Container lifecycle events.

Container Lifecycle Hooks

This page describes how kubelet managed Containers can use the Container lifecycle hook framework to run code triggered by events during their management lifecycle.

Overview

Analogous to many programming language frameworks that have component lifecycle hooks, such as Angular, Kubernetes provides Containers with lifecycle hooks. The hooks enable Containers to be aware of events in their management lifecycle and run code implemented in a handler when the corresponding lifecycle hook is executed.

Container hooks

There are two hooks that are exposed to Containers:

`PostStart`

This hook executes immediately after a container is created. However, there is no guarantee that the hook will execute before the container ENTRYPOINT. No parameters are passed to the handler.

`PreStop`

This hook is called immediately before a container is terminated. It is blocking, meaning it is synchronous, so it must complete before the call to delete the container can be sent. No parameters are passed to the handler.

A more detailed description of the termination behavior can be found in Termination of Pods.

Hook handler implementations

Containers can access a hook by implementing and registering a handler for that hook. There are two types of hook handlers that can be implemented for Containers:

- Exec - Executes a specific command, such as `pre-stop.sh`, inside the cgroups and namespaces of the Container. Resources consumed by the command are counted against the Container.
- HTTP - Executes an HTTP request against a specific endpoint on the Container.

Hook handler execution

When a Container lifecycle management hook is called, the Kubernetes management system executes the handler in the Container registered for that hook.

Hook handler calls are synchronous within the context of the Pod containing the Container. This means that for a `PostStart` hook, the Container ENTRYPOINT and hook fire asynchronously. However, if the hook takes too long to run or hangs, the Container cannot reach a `running` state.

The behavior is similar for a `PreStop` hook. If the hook hangs during execution, the Pod phase stays in a `running` state and never reaches `failed`. If a `PostStart` or `PreStop` hook fails, it kills the Container.

Users should make their hook handlers as lightweight as possible. There are cases, however, when long running commands make sense, such as when saving state prior to stopping a Container.

Hook delivery guarantees

Hook delivery is intended to be *at least once*, which means that a hook may be called multiple times for any given event, such as for `PostStart` or `PreStop`. It is up to the hook implementation to handle this correctly.

Generally, only single deliveries are made. If, for example, an HTTP hook receiver is down and is unable to take traffic, there is no attempt to resend. In some rare cases, however, double delivery may occur. For instance, if a kubelet restarts in the middle of sending a hook, the hook might be resent after the kubelet comes back up.

Debugging Hook handlers

The logs for a Hook handler are not exposed in Pod events. If a handler fails for some reason, it broadcasts an event. For `PostStart`, this is the `FailedPostStartHook` event, and for `PreStop`, this is the `FailedPreStopHook` event. You can see these events by running `kubectl describe pod <pod_name>`. Here is some example output of events from running this command:

```
Events:
  FirstSeen    LastSeen    Count    From                                SubobjectPath
  ---------    --------    -----    ----                                -------------
  1m        1m        1    {default-scheduler }                                   Nor
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  1m        1m        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  38s        38s        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  37s        37s        1    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  38s        37s        2    {kubelet gke-test-cluster-default-pool-a07e5d30-siqd}
  1m        22s         2     {kubelet gke-test-cluster-default-pool-a07e5d30-siq
```

What's next

- Learn more about the Container environment.
- Get hands-on experience attaching handlers to Container lifecycle events.

Pod Overview

This page provides an overview of `Pod`, the smallest deployable object in the Kubernetes object model.

Understanding Pods

A *Pod* is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster.

A Pod encapsulates an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: *a single instance of an application in Kubernetes*, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.
Docker is the most common container runtime used in a Kubernetes Pod, but Pods support other container runtimes as well.
Pods in a Kubernetes cluster can be used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.

- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate “sidecar” container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity.

The Kubernetes Blog has some additional information on Pod use cases. For more information, see:

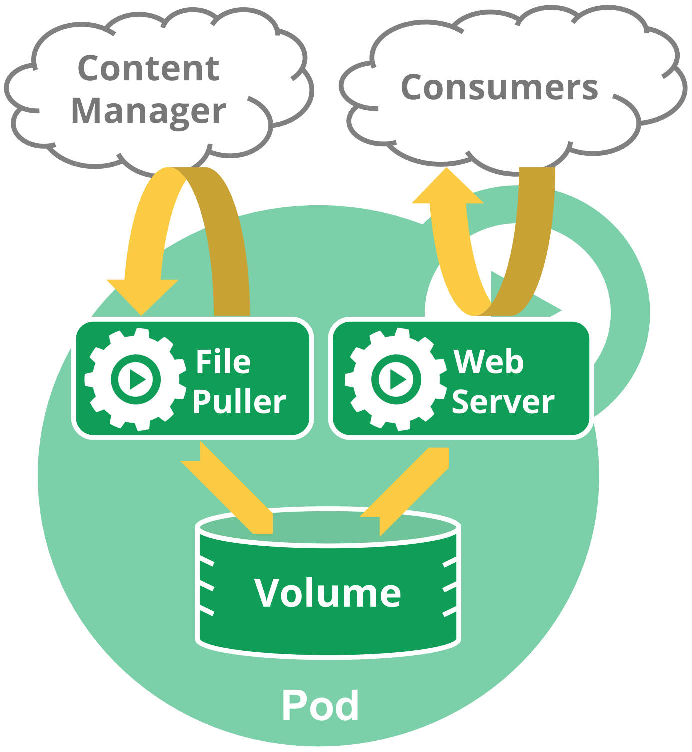
- The Distributed System Toolkit: Patterns for Composite Containers
- Container Design Patterns

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (e.g., run multiple instances), you should use multiple Pods, one for each instance. In Kubernetes, this is generally referred to as *replication*. Replicated Pods are usually created and managed as a group by an abstraction called a Controller. See Pods and Controllers for more information.

How Pods manage multiple Containers

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

Note that grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled. For example, you might have a container that acts as a web server for files in a shared volume, and a separate “sidecar” container that updates those files from a remote source, as in the following diagram:



Pods provide two kinds of shared resources for their constituent containers: *networking* and *storage*.

Networking

Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports. Containers *inside a Pod* can communicate with one another using `localhost`. When containers in a Pod communicate with entities *outside the Pod*, they must coordinate how they use the shared network resources (such as ports).

Storage

A Pod can specify a set of shared storage *volumes*. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See Volumes for more information on how Kubernetes implements shared storage in a Pod.

Working with Pods

You'll rarely create individual Pods directly in Kubernetes–even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a Controller), it is scheduled to run on a Node in your cluster. The Pod remains on that

Node until the process is terminated, the pod object is deleted, or the pod is *evicted* for lack of resources, or the Node fails.

Note: Restarting a container in a Pod should not be confused with restarting the Pod. The Pod itself does not run, but is an environment the containers run in and persists until it is deleted.

Pods do not, by themselves, self-heal. If a Pod is scheduled to a Node that fails, or if the scheduling operation itself fails, the Pod is deleted; likewise, a Pod won't survive an eviction due to a lack of resources or Node maintenance. Kubernetes uses a higher-level abstraction, called a *Controller*, that handles the work of managing the relatively disposable Pod instances. Thus, while it is possible to use Pod directly, it's far more common in Kubernetes to manage your pods using a Controller. See Pods and Controllers for more information on how Kubernetes uses Controllers to implement Pod scaling and healing.

Pods and Controllers

A Controller can create and manage multiple Pods for you, handling replication and rollout and providing self-healing capabilities at cluster scope. For example, if a Node fails, the Controller might automatically replace the Pod by scheduling an identical replacement on a different Node.

Some examples of Controllers that contain one or more pods include:

- Deployment
- StatefulSet
- DaemonSet

In general, Controllers use a Pod Template that you provide to create the Pods for which it is responsible.

Pod Templates

Pod templates are pod specifications which are included in other objects, such as Replication Controllers, Jobs, and DaemonSets. Controllers use Pod Templates to make actual pods.

Rather than specifying the current desired state of all replicas, pod templates are like cookie cutters. Once a cookie has been cut, the cookie has no relationship to the cutter. There is no quantum entanglement. Subsequent changes to the template or even switching to a new template has no direct effect on the pods already created. Similarly, pods created by a replication controller may subsequently be updated directly. This is in deliberate contrast to pods, which do specify the current desired state of all containers belonging to the pod. This approach radically simplifies system semantics and increases the flexibility of the primitive.

What’s next

- Learn more about Pod behavior:
  - Pod Termination
  - Other Pod Topics

Pods

pods are the smallest deployable units of computing that can be created and managed in Kubernetes.

What is a Pod?

A *pod* (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A pod's contents are always co-located and co-scheduled, and run in a shared context. A pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled — in a pre-container world, they would have executed on the same physical or virtual machine.

While Kubernetes supports more container runtimes than just Docker, Docker is the most commonly known runtime, and it helps to describe pods in Docker terms.

The shared context of a pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a pod's context, the individual applications may have further sub-isolations applied.

Containers within a pod share an IP address and port space, and can find each other via `localhost`. They can also communicate with each other using standard inter-process communications like

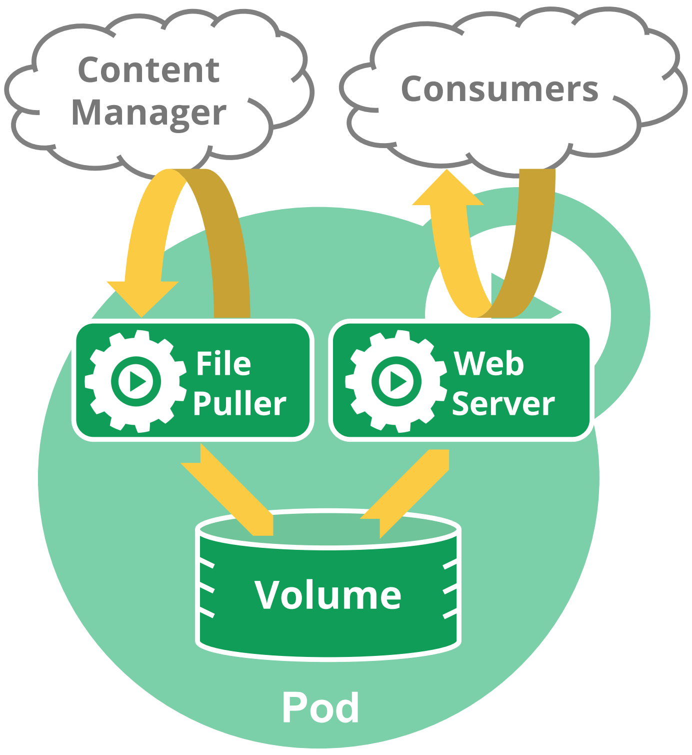
SystemV semaphores or POSIX shared memory. Containers in different pods have distinct IP addresses and can not communicate by IPC.

Applications within a pod also have access to shared volumes, which are defined as part of a pod and are made available to be mounted into each application's filesystem.

In terms of Docker constructs, a pod is modelled as a group of Docker containers with shared namespaces and shared volumes. PID namespace sharing is not yet implemented in Docker.

Like individual application containers, pods are considered to be relatively ephemeral (rather than durable) entities. As discussed in life of a pod, pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion. If a node dies, the pods scheduled to that node are scheduled for deletion, after a timeout period. A given pod (as defined by a UID) is not "rescheduled" to a new node; instead, it can be replaced by an identical pod, with even the same name if desired, but with a new UID (see replication controller for more details). (In the future, a higher-level API may support pod migration.)

When something is said to have the same lifetime as a pod, such as a volume, that means that it exists as long as that pod (with that UID) exists. If that pod is deleted for any reason, even if an identical replacement is created, the related thing (e.g. volume) is also destroyed and created anew.



A multi-container pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Motivation for pods

Management

Pods are a model of the pattern of multiple cooperating processes which form a cohesive unit of service. They simplify application deployment and management by providing a higher-level abstraction than the set of their constituent applications. Pods serve as unit of deployment, horizontal scaling, and replication. Colocation (co-scheduling), shared fate (e.g. termination), coordinated replication, resource sharing, and dependency management are handled automatically for containers in a pod.

Resource sharing and communication

Pods enable data sharing and communication among their constituents.

The applications in a pod all use the same network namespace (same IP and port space), and can thus “find” each other and communicate using `localhost` . Because of this, applications in a pod must coordinate their usage of ports. Each pod has an IP address in a flat shared networking space that has full communication with other physical computers and pods across the network.

The hostname is set to the pod’s Name for the application containers within the pod. More details on networking.

In addition to defining the application containers that run in the pod, the pod specifies a set of shared storage volumes. Volumes enable data to survive container restarts and to be shared among the applications within the pod.

Uses of pods

Pods can be used to host vertically integrated application stacks (e.g. LAMP), but their primary motivation is to support co-located, co-managed helper programs, such as:

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.

- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

Individual pods are not intended to run multiple instances of the same application, in general.

For a longer explanation, see The Distributed System ToolKit: Patterns for Composite Containers.

Alternatives considered

Why not just run multiple programs in a single (Docker) container?

1. Transparency. Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
2. Decoupling software dependencies. The individual containers may be versioned, rebuilt and redeployed independently. Kubernetes may even support live updates of individual containers someday.
3. Ease of use. Users don't need to run their own process managers, worry about signal and exit-code propagation, etc.
4. Efficiency. Because the infrastructure takes on more responsibility, containers can be lighter weight.

Why not support affinity-based co-scheduling of containers?

That approach would provide co-location, but would not provide most of the benefits of pods, such as resource sharing, IPC, guaranteed fate sharing, and simplified management.

Durability of pods (or lack thereof)

Pods aren't intended to be treated as durable entities. They won't survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

In general, users shouldn't need to create pods directly. They should almost always use controllers (e.g., Deployments), even for singletons. Controllers provide self-healing with a cluster scope, as well as replication and rollout management.

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including Borg, Marathon, Aurora, and Tupperware.

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to “proxy” them via controller APIs
- decoupling of pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services — the endpoint controller just watches pods
- clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the “pod controller”
- high-availability applications, which will expect pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions, image prefetching, or live pod migration #3949

There is new first-class support for stateful pods with the StatefulSet controller (currently in beta). The feature was alpha in 1.4 and was called PetSet. For prior versions of Kubernetes, best practice for having stateful pods is to create a replication controller with `replicas` equal to `1` and a corresponding service, see this MySQL deployment example.

Termination of Pods

Because pods represent running processes on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (vs being violently killed with a KILL signal and having no chance to clean up). Users should be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When a user requests deletion of a pod the system records the intended grace period before the pod is allowed to be forcefully killed, and a TERM signal is sent to the main process in each container. Once the grace period has expired the KILL signal is sent to those processes and the pod is then deleted from the API server. If the Kubelet or the container manager is restarted while waiting for processes to terminate, the termination will be retried with the full grace period.

An example flow:

1. User sends command to delete Pod, with default grace period (30s)
2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period.
3. Pod shows up as "Terminating" when listed in client commands
4. (simultaneous with 3) When the Kubelet sees that a Pod has been marked as terminating because the time in 2 has been set, it begins the pod shutdown process.
5. If the pod has defined a preStop hook, it is invoked inside of the pod. If the `preStop` hook is still running after the grace period expires, step 2 is then invoked with a small (2 second) extended grace period.
6. The processes in the Pod are sent the TERM signal.
7. (simultaneous with 3), Pod is removed from endpoints list for service, and are no longer considered part of the set of running pods for replication controllers. Pods that shutdown slowly can continue to serve traffic as load balancers (like the service proxy) remove them from their rotations.
8. When the grace period expires, any processes still running in the Pod are killed with SIGKILL.
9. The Kubelet will finish deleting the Pod on the API server by setting grace period 0 (immediate deletion). The Pod disappears from the API and is no longer visible from the client.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows a user to override the default and specify their own value. The value `0` force deletes the pod. In kubectl version >= 1.5, you must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

Force deletion of pods

Force deletion of a pod is defined as deletion of a pod from the cluster state and etcd immediately. When a force deletion is performed, the apiserver does not wait for confirmation from the kubelet that the pod has been terminated on the node it was running on. It removes the pod in the API immediately so a new pod can be created with the same name. On the node, pods that are set to terminate immediately will still be given a small grace period before being force killed.

Force deletions can be potentially dangerous for some pods and should be performed with caution. In case of StatefulSet pods, please refer to the task documentation for deleting Pods from a StatefulSet.

Privileged mode for pod containers

From Kubernetes v1.1, any container in a pod can enable privileged mode, using the `privileged` flag on the `SecurityContext` of the container spec. This is useful for containers that want to use linux capabilities like manipulating the network stack and accessing devices. Processes within the container get almost the same privileges that are available to processes outside a container. With privileged mode, it should be easier to write network and volume plugins as separate pods that don't need to be compiled into the kubelet.

If the master is running Kubernetes v1.1 or higher, and the nodes are running a version lower than v1.1, then new privileged pods will be accepted by api-server, but will not be launched. They will be pending state. If user calls `kubectl describe pod FooPodName`, user can see the reason why the pod is in pending state. The events table in the describe command output will say:

```
Error validating pod "FooPodName"."FooPodNamespace" from api, ignoring:
spec.containers[0].securityContext.privileged: forbidden '<*>(0xc2089d3248)true'
```

If the master is running a version lower than v1.1, then privileged pods cannot be created. If user attempts to create a pod, that has a privileged container, the user will get the following error:

```
The Pod "FooPodName" is invalid. spec.containers[0].securityContext.privileged:
forbidden '<*>(0xc20b222db0)true'
```

API Object

Pod is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Pod API object.

Pod Lifecycle

This page describes the lifecycle of a Pod.

Pod phase

A Pod's `status` field is a PodStatus object, which has a `phase` field.

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. The phase is not intended to be a comprehensive rollup of observations of Container or Pod state, nor is it intended to be a comprehensive state machine.

The number and meanings of Pod phase values are tightly guarded. Other than what is documented here, nothing should be assumed about Pods that have a given `phase` value.

Here are the possible values for `phase` :

- Pending: The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.
- Running: The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting.

- Succeeded: All Containers in the Pod have terminated in success, and will not be restarted.
- Failed: All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system.
- Unknown: For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions. Each element of the PodCondition array has a `type` field and a `status` field. The `type` field is a string, with possible values PodScheduled, Ready, Initialized, and Unschedulable. The `status` field is a string, with possible values True, False, and Unknown.

Container probes

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the Container. There are three types of handlers:

- ExecAction: Executes a specified command inside the Container. The diagnostic is considered successful if the command exits with a status code of 0.
- TCPSocketAction: Performs a TCP check against the Container's IP address on a specified port. The diagnostic is considered successful if the port is open.
- HTTPGetAction: Performs an HTTP Get request against the Container's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- Success: The Container passed the diagnostic.
- Failure: The Container failed the diagnostic.

- Unknown: The diagnostic failed, so no action should be taken.

The kubelet can optionally perform and react to two kinds of probes on running Containers:

- `livenessProbe` : Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is `Success` .
- `readinessProbe` : Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is `Failure` . If a Container does not provide a readiness probe, the default state is `Success` .

When should you use liveness or readiness probes?

If the process in your Container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod's `restartPolicy` .

If you'd like your Container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a `restartPolicy` of Always or OnFailure.

If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding.

If you want your Container to be able to take itself down for maintenance, you can specify a readiness probe that checks an endpoint specific to readiness that is different from the liveness probe.

Note that if you just want to be able to drain requests when the Pod is deleted, you do not necessarily need a readiness probe; on deletion, the Pod automatically puts itself into an unready state regardless of whether the readiness probe exists. The Pod remains in the unready state while it waits for the Containers in the Pod to stop.

Pod and Container status

For detailed information about Pod Container status, see PodStatus and ContainerStatus. Note that the information reported as Pod status depends on the current ContainerState.

Restart policy

A PodSpec has a `restartPolicy` field with possible values Always, OnFailure, and Never. The default value is Always. `restartPolicy` applies to all Containers in the Pod. `restartPolicy` only refers to restarts of the Containers by the kubelet on the same node. Failed Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution. As discussed in the Pods document, once bound to a node, a Pod will never be rebound to another node.

Pod lifetime

In general, Pods do not disappear until someone destroys them. This might be a human or a controller. The only exception to this rule is that Pods with a `phase` of Succeeded or Failed for more than some duration (determined by the master) will expire and be automatically destroyed.

Three types of controllers are available:

- Use a Job for Pods that are expected to terminate, for example, batch computations. Jobs are appropriate only for Pods with `restartPolicy` equal to OnFailure or Never.
- Use a ReplicationController, ReplicaSet, or Deployment for Pods that are not expected to terminate, for example, web servers. ReplicationControllers are appropriate only for Pods with a `restartPolicy` of Always.
- Use a DaemonSet for Pods that need to run one per machine, because they provide a machine-specific system service.

All three types of controllers contain a PodTemplate. It is recommended to create the appropriate controller and let it create Pods, rather than directly create Pods yourself. That is because Pods alone are not resilient to machine failures, but controllers are.

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the `phase` of all Pods on the lost node to Failed.

Examples

Advanced liveness probe example

Liveness probes are executed by the kubelet, so all requests are made in the kubelet network namespace.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - args:
    - /server
    image: gcr.io/google_containers/liveness
    livenessProbe:
      httpGet:
        # when "host" is not defined, "PodIP" will be used
        # host: my-host
        # when "scheme" is not defined, "HTTP" scheme will be used. Only "HTTP" ar
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
          - name: X-Custom-Header
            value: Awesome
      initialDelaySeconds: 15
      timeoutSeconds: 1
    name: liveness
```

Example states

- Pod is running and has one Container. Container exits with success.
  - Log completion event.

    - If `restartPolicy` is:
        - Always: Restart Container; Pod `phase` stays Running.
        - OnFailure: Pod `phase` becomes Succeeded.
        - Never: Pod `phase` becomes Succeeded.
- Pod is running and has one Container. Container exits with failure.
    - Log failure event.
    - If `restartPolicy` is:
        - Always: Restart Container; Pod `phase` stays Running.
        - OnFailure: Restart Container; Pod `phase` stays Running.
        - Never: Pod `phase` becomes Failed.
- Pod is running and has two Containers. Container 1 exits with failure.
    - Log failure event.
    - If `restartPolicy` is:
        - Always: Restart Container; Pod `phase` stays Running.
        - OnFailure: Restart Container; Pod `phase` stays Running.
        - Never: Do not restart Container; Pod `phase` stays Running.
    - If Container 1 is not running, and Container 2 exits:
        - Log failure event.
        - If `restartPolicy` is:
            - Always: Restart Container; Pod `phase` stays Running.
            - OnFailure: Restart Container; Pod `phase` stays Running.
            - Never: Pod `phase` becomes Failed.
- Pod is running and has one Container. Container runs out of memory.

    - Container terminates in failure.
    - Log OOM event.
    - If `restartPolicy` is:
        - Always: Restart Container; Pod `phase` stays Running.
        - OnFailure: Restart Container; Pod `phase` stays Running.
        - Never: Log failure event; Pod `phase` becomes Failed.
- Pod is running, and a disk dies.
    - Kill all Containers.
    - Log appropriate event.
    - Pod `phase` becomes Failed.
    - If running under a controller, Pod is recreated elsewhere.
- Pod is running, and its node is segmented out.
    - Node controller waits for timeout.
    - Node controller sets Pod `phase` to Failed.
    - If running under a controller, Pod is recreated elsewhere.

What's next

- Get hands-on experience attaching handlers to Container lifecycle events.
- Get hands-on experience configuring liveness and readiness probes.
- Learn more about Container lifecycle hooks.

Init Containers

This feature has exited beta in 1.6. Init Containers can be specified in the PodSpec alongside the app `containers` array. The beta annotation value will still be respected and overrides the PodSpec field value, however, they are deprecated in 1.6 and 1.7. In 1.8, the annotations are no longer supported and must be converted to the PodSpec field.

This page provides an overview of Init Containers, which are specialized Containers that run before app Containers and can contain utilities or setup scripts not present in an app image.

Understanding Init Containers

A Pod can have multiple Containers running apps within it, but it can also have one or more Init Containers, which are run before the app Containers are started.

Init Containers are exactly like regular Containers, except:

- They always run to completion.
- Each one must complete successfully before the next one is started.

If an Init Container fails for a Pod, Kubernetes restarts the Pod repeatedly until the Init Container succeeds. However, if the Pod has a `restartPolicy` of Never, it is not restarted.

To specify a Container as an Init Container, add the `initContainers` field on the PodSpec as a JSON array of objects of type v1.Container alongside the app `containers` array. The status of the init containers is returned in `status.initContainerStatuses` field as an array of the container statuses (similar to the `status.containerStatuses` field).

Differences from regular Containers

Init Containers support all the fields and features of app Containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an Init Container are handled slightly differently, which are documented in Resources below. Also, Init Containers do not support readiness probes because they must run to completion before the Pod can be ready.

If multiple Init Containers are specified for a Pod, those Containers are run one at a time in sequential order. Each must succeed before the next can run. When all of the Init Containers have run to completion, Kubernetes initializes the Pod and runs the application Containers as usual.

What can Init Containers be used for?

Because Init Containers have separate images from app Containers, they have some advantages for start-up related code:

- They can contain and run utilities that are not desirable to include in the app Container image for security reasons.
- They can contain utilities or custom code for setup that is not present in an app image. For example, there is no need to make an image `FROM` another image just to use a tool like `sed`, `awk`, `python`, or `dig` during setup.
- The application image builder and deployer roles can work independently without the need to jointly build a single app image.
- They use Linux namespaces so that they have different filesystem views from app Containers. Consequently, they can be given access to Secrets that app Containers are not able to access.
- They run to completion before any app Containers start, whereas app Containers run in parallel, so Init Containers provide an easy way to block or delay the startup of app Containers until some set of preconditions are met.

Examples

Here are some ideas for how to use Init Containers:

- Wait for a service to be created with a shell command like:

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; exit 1
```

- Register this Pod with a remote server from the downward API with a command like:

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/regist
```

- Wait for some time before starting the app Container with a command like `sleep 60`.
- Clone a git repository into a volume.
- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app Container. For example, place the POD_IP value in a configuration and generate the main app configuration file using Jinja.

More detailed usage examples can be found in the StatefulSets documentation and the Production Pods guide.

Init Containers in use

The following yaml file for Kubernetes 1.5 outlines a simple Pod which has two Init Containers. The first waits for `myservice` and the second waits for `mydb`. Once both containers complete, the Pod will begin.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
        {
            "name": "init-myservice",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup myservice; do echo waiting for
        },
        {
            "name": "init-mydb",
            "image": "busybox",
            "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb
        }
    ]'
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

There is a new syntax in Kubernetes 1.6, although the old annotation syntax still works for 1.6 and 1.7. The new syntax must be used for 1.8 or greater. We have moved the declaration of init containers to `spec`:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice
  - name: init-mydb
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2;
```

1.5 syntax still works on 1.6, but we recommend using 1.6 syntax. In Kubernetes 1.6, Init Containers were made a field in the API. The beta annotation is still respected in 1.6 and 1.7, but is not supported in 1.8 or greater.

Yaml file below outlines the `mydb` and `myservice` services:

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

This Pod can be started and debugged with the following commands:

```
$ kubectl create -f myapp.yaml
pod "myapp-pod" created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
$ kubectl describe -f myapp.yaml
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app=myapp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Events:
  FirstSeen    LastSeen    Count    From                      SubObjectPath
  ---------    --------    -----    ----                      -------------
  16s          16s         1        {default-scheduler }
  16s          16s         1        {kubelet 172.17.4.201}    spec.initContainers{
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{
$ kubectl logs myapp-pod -c init-myservice # Inspect the first init container
$ kubectl logs myapp-pod -c init-mydb      # Inspect the second init container
```

Once we start the **mydb** and **myservice** services, we can see the Init Containers complete and the **myapp-pod** is created:

```
$ kubectl create -f services.yaml
service "myservice" created
service "mydb" created
$ kubectl get -f myapp.yaml
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

This example is very simple but should provide some inspiration for you to create your own Init Containers.

Detailed behavior

During the startup of a Pod, the Init Containers are started in order, after the network and volumes are initialized. Each Container must exit successfully before the next is started. If a Container fails to start due to the runtime or exits with failure, it is retried according to the Pod `restartPolicy`. However, if the Pod `restartPolicy` is set to Always, the Init Containers use `RestartPolicy` OnFailure.

A Pod cannot be `Ready` until all Init Containers have succeeded. The ports on an Init Container are not aggregated under a service. A Pod that is initializing is in the `Pending` state but should have a condition `Initializing` set to true.

If the Pod is restarted, all Init Containers must execute again.

Changes to the Init Container spec are limited to the container image field. Altering an Init Container image field is equivalent to restarting the Pod.

Because Init Containers can be restarted, retried, or re-executed, Init Container code should be idempotent. In particular, code that writes to files on `EmptyDirs` should be prepared for the possibility that an output file already exists.

Init Containers have all of the fields of an app Container. However, Kubernetes prohibits `readinessProbe` from being used because Init Containers cannot define readiness distinct from completion. This is enforced during validation.

Use `activeDeadlineSeconds` on the Pod and `livenessProbe` on the Container to prevent Init Containers from failing forever. The active deadline includes Init Containers.

The name of each app and Init Container in a Pod must be unique; a validation error is thrown for any Container sharing a name with another.

Resources

Given the ordering and execution for Init Containers, the following rules for resource usage apply:

- The highest of any particular resource request or limit defined on all Init Containers is the *effective init request/limit*
- The Pod's *effective request/limit* for a resource is the higher of:
  - the sum of all app Containers request/limit for a resource
  - the effective init request/limit for a resource
- Scheduling is done based on effective requests/limits, which means Init Containers can reserve resources for initialization that are not used during the life of the Pod.
- QoS tier of the Pod's *effective QoS tier* is the QoS tier for Init Containers and app containers alike.

Quota and limits are applied based on the effective Pod request and limit.

Pod level cgroups are based on the effective Pod request and limit, the same as the scheduler.

Pod restart reasons

A Pod can restart, causing re-execution of Init Containers, for the following reasons:

- A user updates the PodSpec causing the Init Container image to change. App Container image changes only restart the app Container.
- The Pod infrastructure container is restarted. This is uncommon and would have to be done by someone with root access to nodes.
- All containers in a Pod are terminated while `restartPolicy` is set to Always, forcing a restart, and the Init Container completion record has been lost due to garbage collection.

Support and compatibility

A cluster with Apiserver version 1.6.0 or greater supports Init Containers using the `spec.initContainers` field. Previous versions support Init Containers using the alpha or beta annotations. The `spec.initContainers` field is also mirrored into alpha and beta annotations so that Kubelets version 1.3.0 or greater can execute Init Containers, and so that a version 1.6 apiserver can safely be rolled back to version 1.5.x without losing Init Container functionality for existing created pods.

In Apiserver and Kubelet versions 1.8.0 or greater, support for the alpha and beta annotations is removed, requiring a conversion from the deprecated annotations to the `spec.initContainers` field.

What's next

- Creating a Pod that has an Init Container

Disruptions

This guide is for application owners who want to build highly available applications, and thus need to understand what types of Disruptions can happen to Pods.

It is also for Cluster Administrators who want to perform automated cluster actions, like upgrading and autoscaling clusters.

Voluntary and Involuntary Disruptions

Pods do not disappear until someone (a person or a controller) destroys them, or there is an unavoidable hardware or system software error.

We call these unavoidable cases *involuntary disruptions* to an application. Examples are:

- a hardware failure of the physical machine backing the node
- cluster administrator deletes VM (instance) by mistake
- cloud provider or hypervisor failure makes VM disappear
- a kernel panic
- the node disappears from the cluster due to cluster network partition
- eviction of a pod due to the node being out-of-resources.

Except for the out-of-resources condition, all these conditions should be familiar to most users; they are not specific to Kubernetes.

We call other cases *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical application owner actions include:

- deleting the deployment or other controller that manages the pod
- updating a deployment's pod template causing a restart
- directly deleting a pod (e.g. by accident)

Cluster Administrator actions include:

- Draining a node for repair or upgrade.
- Draining a node from a cluster to scale the cluster down (learn about Cluster Autoscaling).
- Removing a pod from a node to permit something else to fit on that node.

These actions might be taken directly by the cluster administrator, or by automation run by the cluster administrator, or by your cluster hosting provider.

Ask your cluster administrator or consult your cloud provider or distribution documentation to determine if any sources of voluntary disruptions are enabled for your cluster. If none are enabled, you can skip creating Pod Disruption Budgets.

Dealing with Disruptions

Here are some ways to mitigate involuntary disruptions:

- Ensure your pod requests the resources it needs.
- Replicate your application if you need higher availability. (Learn about running replicated stateless and stateful applications.)
- For even higher availability when running replicated applications, spread applications across racks (using anti-affinity) or across zones (if using a multi-zone cluster.)

The frequency of voluntary disruptions varies. On a basic Kubernetes cluster, there are no voluntary disruptions at all. However, your cluster administrator or hosting provider may run some additional services which cause voluntary disruptions. For example, rolling out node software updates can cause voluntary updates. Also, some implementations of cluster (node) autoscaling may cause voluntary disruptions to defragment and compact nodes. You cluster administrator or hosting provider should have documented what level of voluntary disruptions, if any, to expect.

Kubernetes offers features to help run highly available applications at the same time as frequent voluntary disruptions. We call this set of features *Disruption Budgets*.

How Disruption Budgets Work

An Application Owner can create a `PodDisruptionBudget` object (PDB) for each application. A PDB limits the number pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

Cluster managers and hosting providers should use tools which respect Pod Disruption Budgets by calling the Eviction API instead of directly deleting pods. Examples are the `kubectl drain` command and the Kubernetes-on-GCE cluster upgrade script (`cluster/gce/upgrade.sh`).

When a cluster administrator wants to drain a node they use the `kubectl drain` command. That tool tries to evict all the pods on the machine. The eviction request may be temporarily rejected, and the tool periodically retries all failed requests until all pods are terminated, or until a configurable timeout is reached.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment which has a `spec.replicas: 5` is supposed to have 5 pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one, but not two pods, at a time.

The group of pods that comprise the application is specified using a label selector, the same as the one used by the application's controller (deployment, stateful-set, etc).

The "intended" number of pods is computed from the `.spec.replicas` of the pods controller. The controller is discovered from the pods using the `.metadata.ownerReferences` of the object.

PDBs cannot prevent involuntary disruptions from occurring, but they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but controllers (like deployment and stateful-set) are not limited by PDBs when doing rolling upgrades – the handling of failures during application updates is configured in the controller spec. (Learn about updating a deployment.)

When a pod is evicted using the eviction API, it is gracefully terminated (see `terminationGracePeriodSeconds` in PodSpec.)

PDB Example

Consider a cluster with 3 nodes, `node-1` through `node-3`. The cluster is running several applications. One of them has 3 replicas initially called `pod-a`, `pod-b`, and `pod-c`. Another, unrelated pod without a PDB, called `pod-x`, is also shown. Initially, the pods are laid out as follows:

node-1	node-2	node-3
pod-a *available*	pod-b *available*	pod-c *available*
pod-x *available*		

All 3 pods are part of a deployment, and they collectively have a PDB which requires there be at least 2 of the 3 pods to be available at all times.

For example, assume the cluster administrator wants to reboot into a new kernel version to fix a bug in the kernel. The cluster administrator first tries to drain `node-1` using the `kubectl drain` command. That tool tries to evict `pod-a` and `pod-x`. This succeeds immediately. Both pods go into the `terminating` state at the same time. This puts the cluster in this state:

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*		

The deployment notices that one of the pods is terminating, so it creates a replacement called `pod-d`. Since `node-1` is cordoned, it lands on another node. Something has also created `pod-y` as a replacement for `pod-x`.

(Note: for a StatefulSet, `pod-a`, which would be called something like `pod-1`, would need to terminate completely before its replacement, which is also called `pod-1` but has a different UID, could be created. Otherwise, the example applies to a StatefulSet as well.)

Now the cluster is in this state:

node-1 *draining*	node-2	node-3
pod-a *terminating*	pod-b *available*	pod-c *available*
pod-x *terminating*	pod-d *starting*	pod-y

At some point, the pods terminate, and the cluster looks like this:

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *starting*	pod-y

At this point, if an impatient cluster administrator tries to drain `node-2` or `node-3`, the drain command will block, because there are only 2 available pods for the deployment, and its PDB requires at least 2. After some time passes, `pod-d` becomes available.

The cluster state now looks like this:

node-1 *drained*	node-2	node-3
	pod-b *available*	pod-c *available*
	pod-d *available*	pod-y

Now, the cluster administrator tries to drain `node-2`. The drain command will try to evict the two pods in some order, say `pod-b` first and then `pod-d`. It will succeed at evicting `pod-b`. But, when it tries to evict `pod-d`, it will be refused because that would leave only one pod available for the deployment.

The deployment creates a replacement for `pod-b` called `pod-e`. However, not there are not enough resources in the cluster to schedule `pod-e`. So, the drain will again block. The cluster may end up in this state:

node-1 *drained*	node-2	node-3	*no node*
	pod-b *available*	pod-c *available*	pod-e *pending*
	pod-d *available*	pod-y	

At this point, the cluster administrator needs to add a node back to the cluster to proceed with the upgrade.

You can see how Kubernetes varies the rate at which disruptions can happen, according to:

- how many replicas an application needs
- how long it takes to gracefully shutdown an instance
- how long it takes a new instance to start up
- the type of controller
- the cluster’s resource capacity

Separating Cluster Owner and Application Owner Roles

Often, it is useful to think of the Cluster Manager and Application Owner as separate roles with limited knowledge of each other. This separation of responsibilities may make sense in these scenarios:

- when there are many application teams sharing a Kubernetes cluster, and there is natural specialization of roles
- when third-party tools or services are used to automate cluster management

Pod Disruption Budgets support this separation of roles by providing an interface between the roles.

If you do not have such a separation of responsibilities in your organization, you may not need to use Pod Disruption Budgets.

How to perform Disruptive Actions on your Cluster

If you are a Cluster Administrator, and you need to perform a disruptive action on all the nodes in your cluster, such as a node or system software upgrade, here are some options:

- Accept downtime during the upgrade.
- Fail over to another complete replica cluster.
  - No downtime, but may be costly both for the duplicated nodes, and for human effort to orchestrate the switchover.
- Write disruption tolerant applications and use PDBs.
  - No downtime.
  - Minimal resource duplication.
  - Allows more automation of cluster administration.
  - Writing disruption-tolerant applications is tricky, but the work to tolerate voluntary disruptions largely overlaps with work to support autoscaling and tolerating involuntary disruptions.

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.
- Learn more about draining nodes

Replica Sets

ReplicaSet is the next-generation Replication Controller. The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector support. ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

How to use a ReplicaSet

Most `kubectl` commands that support Replication Controllers also support ReplicaSets. One exception is the `rolling-update` command. If you want the rolling update functionality please consider using Deployments instead. Also, the `rolling-update` command is imperative whereas Deployments are declarative, so we recommend using Deployments through the `rollout` command.

While ReplicaSets can be used independently, today it's mainly used by Deployments as a mechanism to orchestrate pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets.

When to use a ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

Example

frontend.yaml

```
apiVersion: apps/v1beta2 # for versions before 1.6.0 use extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
        ports:
        - containerPort: 80
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster should create the defined ReplicaSet and the pods that it manages.

```
$ kubectl create -f frontend.yaml
replicaset "frontend" created
$ kubectl describe rs/frontend
Name:           frontend
Namespace:      default
Selector:       tier=frontend,tier in (frontend)
Labels:         app=guestbook
                tier=frontend
Annotations:    <none>
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=guestbook
                tier=frontend
  Containers:
   php-redis:
    Image:      gcr.io/google_samples/gb-frontend:v3
    Port:       80/TCP
    Requests:
      cpu:      100m
      memory:   100Mi
    Environment:
      GET_HOSTS_FROM:   dns
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From                    SubobjectPath    Type
  ---------    --------    -----    ----                    -------------    --------
  1m           1m          1        {replicaset-controller }                 Normal
  1m           1m          1        {replicaset-controller }                 Normal
  1m           1m          1        {replicaset-controller }                 Normal
$ kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
frontend-9si5l   1/1       Running   0          1m
frontend-dnjpy   1/1       Running   0          1m
frontend-qhloh   1/1       Running   0          1m
```

Writing a ReplicaSet Spec

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For general information about working with manifests, see here, here, and here.

A ReplicaSet also needs a `.spec` section.

Pod Template

The `.spec.template` is the only required field of the `.spec`. The `.spec.template` is a pod template. It has exactly the same schema as a pod, except that it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields of a pod, a pod template in a ReplicaSet must specify appropriate labels and an appropriate restart policy.

For labels, make sure to not overlap with other controllers. For more information, see pod selector.

For restart policy, the only allowed value for `.spec.template.spec.restartPolicy` is `Always`, which is the default.

For local container restarts, ReplicaSet delegates to an agent on the node, for example the Kubelet or Docker.

Pod Selector

The `.spec.selector` field is a label selector. A ReplicaSet manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicaSet to be replaced without affecting the running pods.

The `.spec.template.metadata.labels` must match the `.spec.selector`, or it will be rejected by the API.

In Kubernetes 1.8 the API version `apps/v1beta2` on the ReplicaSet kind is the current version and is enabled by default. The API version `extensions/v1beta1` is deprecated. In API version `apps/v1beta2`, `.spec.selector` and `.metadata.labels` no longer default to `.spec.template.metadata.labels` if not set. So they must be set explicitly. Also note that `.spec.selector` is immutable after creation starting in API version `apps/v1beta2`.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicaSet, or with another controller such as a Deployment. If you do so, the ReplicaSet thinks that it created the other pods. Kubernetes does not stop you from doing this.

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself.

Labels on a ReplicaSet

The ReplicaSet can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels` . However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicaSet.

Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas` . The number running at any time may be higher or lower, such as if the replicas were just increased or decreased, or if a pod is gracefully shut down, and a replacement starts early.

If you do not specify `.spec.replicas` , then it defaults to 1.

Working with ReplicaSets

Deleting a ReplicaSet and its Pods

To delete a ReplicaSet and all its pods, use `kubectl delete` . Kubectl will scale the ReplicaSet to zero and wait for it to delete each pod before deleting the ReplicaSet itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or go client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicaSet).

Deleting just a ReplicaSet

You can delete a ReplicaSet without affecting any of its pods, using `kubectl delete` with the `--cascade=false` option.

When using the REST API or go client library, simply delete the ReplicaSet object.

Once the original is deleted, you can create a new ReplicaSet to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old pods. However, it will not make any effort to make existing pods match a new, different pod template. To update pods to a new spec in a controlled way, use a rolling update.

Isolating pods from a ReplicaSet

Pods may be removed from a ReplicaSet's target set by changing their labels. This technique may be used to remove pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Scaling a ReplicaSet

A ReplicaSet can be easily scaled up or down by simply updating the `.spec.replicas` field. The ReplicaSet controller ensures that that a desired number of pods with a matching label selector are available and operational.

ReplicaSet as an Horizontal Pod Autoscaler Target

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example HPA targeting the ReplicaSet we created in the previous example.

hpa-rs.yaml

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Saving this manifest into `hpa-rs.yaml` and submitting it to a Kubernetes cluster should create the defined HPA that autoscales the target ReplicaSet depending on the CPU usage of the replicated pods.

```
kubectl create -f hpa-rs.yaml
```

Alternatively, you can use the `kubectl autoscale` command to accomplish the same (and it's easier!)

```
kubectl autoscale rs frontend
```

Alternatives to ReplicaSet

Deployment (Recommended)

[`Deployment`] is a higher-level API object that updates its underlying ReplicaSets and their Pods in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features. For more information on running a stateless application using a Deployment, please read Run a Stateless Application Using a Deployment.

Bare Pods

Unlike the case where a user directly created pods, a ReplicaSet replaces pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicaSet even if your application requires only a single pod. Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node. A ReplicaSet delegates local container restarts to some agent on the node (for example, Kubelet or Docker).

Job

Use a `Job` instead of a ReplicaSet for pods that are expected to terminate on their own (that is, batch jobs).

DaemonSet

Use a `DaemonSet` instead of a ReplicaSet for pods that provide a machine-level function, such as machine monitoring or machine logging. These pods have a lifetime that is tied to a machine lifetime: the pod needs to be running on the machine before other pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

Replication Controller

NOTE: A `Deployment` that configures a `ReplicaSet` is now the recommended way to set up replication.

A *ReplicationController* ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

How a ReplicationController Works

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to “rc” or “rcs” in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers.

Running an example ReplicationController

This example ReplicationController config runs three copies of the nginx web server.

replication.yaml

replication.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Run the example job by downloading the example file and then running this command:

```
$ kubectl create -f ./replication.yaml
replicationcontroller "nginx" created
```

Check on the status of the ReplicationController using this command:

```
$ kubectl describe replicationcontrollers/nginx
Name:        nginx
Namespace:   default
Selector:    app=nginx
Labels:      app=nginx
Annotations:    <none>
Replicas:    3 current / 3 desired
Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen       LastSeen     Count    From                        SubobjectPath
  ---------       --------     -----    ----                        -------------
  20s             20s          1        {replication-controller }
  20s             20s          1        {replication-controller }
  20s             20s          1        {replication-controller }
```

Here, three pods are created, but none is running yet, perhaps because the image is being pulled. A little later, the same command may show:

```
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

To list all the pods that belong to the ReplicationController in a machine readable form, you can use a command like this:

```
$ pods=$(kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata
echo $pods
nginx-3ntk0 nginx-4ok8v nginx-qrm3m
```

Here, the selector is the same as the selector for the ReplicationController (seen in the `kubectl describe` output, and in a different form in `replication.yaml`. The `--output=jsonpath` option specifies an expression that just gets the name from each pod in the returned list.

Writing a ReplicationController Spec

As with all other Kubernetes config, a ReplicationController needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see here, here, and here.

A ReplicationController also needs a `.spec` section.

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a ReplicationController must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See pod selector.

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

For local container restarts, ReplicationControllers delegate to an agent on the node, for example the Kubelet or Docker.

Labels on the ReplicationController

The ReplicationController can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`; if `.metadata.labels` is not specified then it defaults to `.spec.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicationController.

Pod Selector

The `.spec.selector` field is a label selector. A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

If specified, the `.spec.template.metadata.labels` must be equal to the `.spec.selector`, or it will be rejected by the API. If `.spec.selector` is unspecified, it will be defaulted to `.spec.template.metadata.labels`.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicationController, or with another controller such as Job. If you do so, the ReplicationController thinks that it created the other pods. Kubernetes does not stop you from doing this.

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself (see below).

Multiple Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas` to the number of pods you would like to have running concurrently. The number running at any time may be higher or lower, such as if the replicas were just increased or decreased, or if a pod is gracefully shutdown, and a replacement starts early.

If you do not specify `.spec.replicas`, then it defaults to 1.

Working with ReplicationControllers

Deleting a ReplicationController and its Pods

To delete a ReplicationController and all its pods, use `kubectl delete`. Kubectl will scale the ReplicationController to zero and wait for it to delete each pod before deleting the ReplicationController itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or go client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicationController).

Deleting just a ReplicationController

You can delete a ReplicationController without affecting any of its pods.

Using kubectl, specify the `--cascade=false` option to `kubectl delete`.

When using the REST API or go client library, simply delete the ReplicationController object.

Once the original is deleted, you can create a new ReplicationController to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old pods. However, it will not make any effort to make existing pods match a new, different pod template. To update pods to a new spec in a controlled way, use a rolling update.

Isolating pods from a ReplicationController

Pods may be removed from a ReplicationController's target set by changing their labels. This technique may be used to remove pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Common usage patterns

Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, a ReplicationController will ensure that the specified number of pods exists, even in the event of node failure or pod termination (for example, due to an action by another control agent).

Scaling

The ReplicationController makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the `replicas` field.

Rolling updates

The ReplicationController is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in #1353, the recommended approach is to create a new ReplicationController with 1 replica, scale the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two ReplicationControllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

Rolling update is implemented in the client tool `kubectl rolling-update`. Visit `kubectl rolling-update` task for more concrete examples.

Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with `tier in (frontend), environment in (prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a ReplicationController with `replicas` set to 9 for the bulk of the replicas, with labels `tier=frontend, environment=prod, track=stable`, and another ReplicationController with `replicas` set to 1 for the canary, with labels `tier=frontend, environment=prod, track=canary`. Now the service is covering both the canary and non-canary pods. But you can mess with the ReplicationControllers separately to test things out, monitor the results, etc.

Using ReplicationControllers with Services

Multiple ReplicationControllers can sit behind a single service, so that, for example, some traffic goes to the old version, and some goes to the new version.

A ReplicationController will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be composed of pods controlled by multiple ReplicationControllers, and it is expected that many ReplicationControllers may be created and destroyed over the lifetime of a service (for instance, to perform an update of pods that run the service). Both services themselves and their clients should remain oblivious to the ReplicationControllers that maintain the pods of the services.

Writing programs for Replication

Pods created by a ReplicationController are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but ReplicationControllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the etcd lock module or RabbitMQ work queues, as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (for example, cpu or memory), should be performed by another online controller process, not unlike the ReplicationController itself.

Responsibilities of the ReplicationController

The ReplicationController simply ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, readiness and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The ReplicationController is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in #492), which would change its `replicas` field. We will not add scheduling policies (for example, spreading) to the ReplicationController. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsewhere. We even plan to factor out the mechanism for bulk pod creation (#170).

The ReplicationController is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The “macro” operations currently supported by kubectl (run, stop, scale, rolling-update) are proof-of-concept examples of this. For instance, we could imagine something like Asgard managing ReplicationControllers, auto-scalers, services, scheduling policies, canaries, etc.

API Object

Replication controller is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: ReplicationController API object.

Alternatives to ReplicationController

ReplicaSet

`ReplicaSet` is the next-generation ReplicationController that supports the new set-based label selector. It's mainly used by `Deployment` as a mechanism to orchestrate pod creation, deletion and updates. Note that we recommend using Deployments instead of directly using Replica Sets, unless you require custom update orchestration or don't require updates at all.

Deployment (Recommended)

`Deployment` is a higher-level API object that updates its underlying Replica Sets and their Pods in a similar fashion as `kubectl rolling-update`. Deployments are recommended if you want this rolling update functionality, because unlike `kubectl rolling-update`, they are declarative, server-side, and have additional features.

Bare Pods

Unlike in the case where a user directly created pods, a ReplicationController replaces pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicationController even if your application requires only a single pod. Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node. A ReplicationController delegates local container restarts to some agent on the node (for example, Kubelet or Docker).

Job

Use a `Job` instead of a ReplicationController for pods that are expected to terminate on their own (that is, batch jobs).

DaemonSet

Use a `DaemonSet` instead of a ReplicationController for pods that provide a machine-level function, such as machine monitoring or machine logging. These pods have a lifetime that is tied to a machine lifetime: the pod needs to be running on the machine before other pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

For more information

Read Run Stateless AP Replication Controller.

Deployments

A *Deployment* controller provides declarative updates for Pods and ReplicaSets.

You describe a *desired state* in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

> **Note:** You should not manage ReplicaSets owned by a Deployment. All the use cases should be covered by manipulating the Deployment object. Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

Use Case

The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load.
- Pause the Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore.

Creating a Deployment

The following is an example of a Deployment. It creates a ReplicaSet to bring up three `nginx` Pods:

nginx-deployment.yaml

```yaml
apiVersion: apps/v1beta2 # for versions before 1.7.0 use apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

In this example:

- A Deployment named `nginx-deployment` is created, indicated by the `metadata: name` field.
- The Deployment creates three replicated Pods, indicated by the `replicas` field.
- The Pod template's specification, or `template: spec` field, indicates that the Pods run one container, `nginx`, which runs the `nginx` Docker Hub image at version 1.7.9.
- The Deployment opens port 80 for use by the Pods.

The `template` field contains the following instructions:

- The Pods are labeled `app: nginx`
- Create one container and name it `nginx`.

- Run the `nginx` image at version `1.7.9`.
- Open port `80` so that the container can send and accept traffic.

To create this Deployment, run the following command:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes.github.i
```

Note: You can append `--record` to this command to record the current command in the annotations of the created or updated resource. This is useful for future review, such as investigating which commands were executed in each Deployment revision.

Next, run `kubectl get deployments`. The output is similar to the following:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         0         0            0           1s
```

When you inspect the Deployments in your cluster, the following fields are displayed:

- `NAME` lists the names of the Deployments in the cluster.
- `DESIRED` displays the desired number of *replicas* of the application, which you define when you create the Deployment. This is the *desired state*.
- `CURRENT` displays how many replicas are currently running.
- `UP-TO-DATE` displays the number of replicas that have been updated to achieve the desired state.
- `AVAILABLE` displays how many replicas of the application are available to your users.
- `AGE` displays the amount of time that the application has been running.

Notice how the values in each field correspond to the values in the Deployment specification:

- The number of desired replicas is 3 according to `spec: replicas` field.
- The number of current replicas is 0 according to the `.status.replicas` field.

- The number of up-to-date replicas is 0 accoridng to the `.status.updatedReplicas` field.
- The number of available replicas is 0 according to the `.status.availableReplicas` field.

To see the Deployment rollout status, run `kubectl rollout status deployment/nginx-deployment`. This command returns the following output:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

Run the `kubectl get deployments` again a few seconds later:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

Notice that the Deployment has created all three replicas, and all replicas are up-to-date (they contain the latest Pod template) and available (the Pod status is Ready for at least the value of the Deployment's `.spec.minReadySeconds` field).

To see the ReplicaSet (`rs`) created by the deployment, run `kubectl get rs`:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-2035384211   3         3         3       18s
```

Notice that the name of the ReplicaSet is always formatted as `[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]`. The hash value is automatically generated when the Deployment is created.

To see the labels automatically generated for each pod, run `kubectl get pods --show-labels`. The following output is returned:

```
NAME                                READY     STATUS    RESTARTS   AGE       LABEL
nginx-deployment-2035384211-7ci7o   1/1       Running   0          18s       app=n
nginx-deployment-2035384211-kzszj   1/1       Running   0          18s       app=n
nginx-deployment-2035384211-qqcnn   1/1       Running   0          18s       app=n
```

The created ReplicaSet ensures that there are three `nginx` Pods running at all times.

> **Note:** You must specify an appropriate selector and Pod template labels in a Deployment (in this case, `app: nginx`). Do not overlap labels or selectors with other controllers (including other Deployments and StatefulSets). Kubernetes doesn't stop you from overlapping, and if multiple controllers have overlapping selectors those controllers might conflict and behave unexpectedly.

Pod-template-hash label

> **Note:** Do not change this label.

The `pod-template-hash label` is added by the Deployment controller to every ReplicaSet that a Deployment creates or adopts.

This label ensures that child ReplicaSets of a Deployment do not overlap. It is generated by hashing the `PodTemplate` of the ReplicaSet and using the resulting hash as the label value that is added to the ReplicaSet selector, Pod template labels, and in any existing Pods that the ReplicaSet might have.

Updating a Deployment

> **Note:** A Deployment's rollout is triggered if and only if the Deployment's pod template (that is, `.spec.template`) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

Suppose that we now want to update the nginx Pods to use the `nginx:1.9.1` image instead of the `nginx:1.7.9` image.

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

Alternatively, we can `edit` the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.7.9` to `nginx:1.9.1`:

```
$ kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
```

To see the rollout status, run:

```
$ kubectl rollout status deployment/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

After the rollout succeeds, you may want to `get` the Deployment:

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           36s
```

The number of up-to-date replicas indicates that the Deployment has updated the replicas to the latest configuration. The current replicas indicates the total replicas this Deployment manages, and the available replicas indicates the number of current replicas that are available.

We can run `kubectl get rs` to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         3       6s
nginx-deployment-2035384211   0         0         0       36s
```

Running `get pods` should now show only the new Pods:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1564180365-khku8   1/1       Running   0          14s
nginx-deployment-1564180365-nacti   1/1       Running   0          14s
nginx-deployment-1564180365-z9gth   1/1       Running   0          14s
```

Next time we want to update these Pods, we only need to update the Deployment's pod template again.

Deployment can ensure that only a certain number of Pods may be down while they are being updated. By default, it ensures that at least 1 less than the desired number of Pods are up (1 max unavailable).

Deployment can also ensure that only a certain number of Pods may be created above the desired number of Pods. By default, it ensures that at most 1 more than the desired number of Pods are up (1 max surge).

In a future version of Kubernetes, the defaults will change from 1-1 to 25%-25%.

For example, if you look at the above Deployment closely, you will see that it first created a new Pod, then deleted some old Pods and created new ones. It does not kill old Pods until a sufficient number of new Pods have come up, and does not create new Pods until a sufficient number of old Pods have been killed. It makes sure that number of available Pods is at least 2 and the number of total Pods is at most 4.

```
$ kubectl describe deployments
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 12:01:06 -0700
Labels:         app=nginx
Annotations:    deployment.kubernetes.io/revision=2
Selector:       app=nginx
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx:1.9.1
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                     SubobjectPath   Type
  --------- --------    -----   ----                     -------------   --------
  36s       36s         1       {deployment-controller }                 Normal
  23s       23s         1       {deployment-controller }                 Normal
  23s       23s         1       {deployment-controller }                 Normal
  23s       23s         1       {deployment-controller }                 Normal
  21s       21s         1       {deployment-controller }                 Normal
  21s       21s         1       {deployment-controller }                 Normal
```

Here we see that when we first created the Deployment, it created a ReplicaSet (nginx-deployment-2035384211) and scaled it up to 3 replicas directly. When we updated the Deployment, it created a new ReplicaSet (nginx-deployment-1564180365) and scaled it up to 1 and then scaled down the old ReplicaSet to 2, so that at least 2 Pods were available and at most 4 Pods were created at all times. It then continued scaling up and down the new and the old ReplicaSet, with the same rolling update strategy. Finally, we'll have 3 available replicas in the new ReplicaSet, and the old ReplicaSet is scaled down to 0.

Rollover (aka multiple updates in-flight)

Each time a new deployment object is observed by the deployment controller, a ReplicaSet is created to bring up the desired Pods if there is no existing ReplicaSet doing so. Existing ReplicaSet controlling Pods whose labels match `.spec.selector` but whose template does not match `.spec.template` are scaled down. Eventually, the new ReplicaSet will be scaled to `.spec.replicas` and all old ReplicaSets will be scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment will create a new ReplicaSet as per the update and start scaling that up, and will roll over the ReplicaSet that it was scaling up previously – it will add it to its list of old ReplicaSets and will start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of `nginx:1.7.9`, but then updates the Deployment to create 5 replicas of `nginx:1.9.1`, when only 3 replicas of `nginx:1.7.9` had been created. In that case, Deployment will immediately start killing the 3 `nginx:1.7.9` Pods that it had created, and will start creating `nginx:1.9.1` Pods. It will not wait for 5 replicas of `nginx:1.7.9` to be created before changing course.

Label selector updates

It is generally discouraged to make label selector updates and it is suggested to plan your selectors up front. In any case, if you need to perform a label selector update, exercise great caution and make sure you have grasped all of the implications.

> **Note:** In API version `apps/v1beta2`, a Deployment's label selector is immutable after it gets created.

- Selector additions require the pod template labels in the Deployment spec to be updated with the new label too, otherwise a validation error is returned. This change is a non-overlapping one, meaning that the new selector does not select ReplicaSets and Pods created with the old selector, resulting in orphaning all old ReplicaSets and creating a new ReplicaSet.
- Selector updates – that is, changing the existing value in a selector key – result in the same behavior as additions.

- Selector removals – that is, removing an existing key from the Deployment selector – do not require any changes in the pod template labels. No existing ReplicaSet is orphaned, and a new ReplicaSet is not created, but note that the removed label still exists in any existing Pods and ReplicaSets.

Rolling Back a Deployment

Sometimes you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want (you can change that by modifying revision history limit).

> **Note:** A Deployment's revision is created when a Deployment's rollout is triggered. This means that the new revision is created if and only if the Deployment's pod template (`.spec.template`) is changed, for example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision, so that we can facilitate simultaneous manual- or auto-scaling. This means that when you roll back to an earlier revision, only the Deployment's pod template part is rolled back.

Suppose that we made a typo while updating the Deployment, by putting the image name as `nginx:1.91` instead of `nginx:1.9.1` :

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
deployment "nginx-deployment" image updated
```

The rollout will be stuck.

```
$ kubectl rollout status deployments nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

Press Ctrl-C to stop the above rollout status watch. For more information on stuck rollouts, read more here.

You will also see that both the number of old replicas (nginx-deployment-1564180365 and nginx-deployment-2035384211) and new replicas (nginx-deployment-3066724191) are 2.

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   2         2         0       25s
nginx-deployment-2035384211   0         0         0       36s
nginx-deployment-3066724191   2         2         2       6s
```

Looking at the Pods created, you will see that the 2 Pods created by new ReplicaSet are stuck in an image pull loop.

```
$ kubectl get pods
NAME                                READY     STATUS             RESTARTS   AGE
nginx-deployment-1564180365-70iae   1/1       Running            0          25s
nginx-deployment-1564180365-jbqqo   1/1       Running            0          25s
nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff   0          6s
nginx-deployment-3066724191-eocby   0/1       ImagePullBackOff   0          6s
```

Note: The Deployment controller will stop the bad rollout automatically, and will stop scaling up the new ReplicaSet. This depends on the rollingUpdate parameters (`maxUnavailable` specifically) that you have specified. Kubernetes by default sets the value to 1 and spec.replicas to 1 so if you haven't cared about setting those parameters, your Deployment can have 100% unavailability by default! This will be fixed in Kubernetes in a future version.

```
$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       2 updated | 3 total | 2 available | 2 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     nginx-deployment-1564180365 (2/2 replicas created)
NewReplicaSet:      nginx-deployment-3066724191 (2/2 replicas created)
Events:
  FirstSeen LastSeen    Count   From                    SubobjectPath   Type
  --------- --------    -----   ----                    -------------   --------
  1m        1m          1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  22s       22s         1       {deployment-controller }                Normal
  21s       21s         1       {deployment-controller }                Normal
  21s       21s         1       {deployment-controller }                Normal
  13s       13s         1       {deployment-controller }                Normal
  13s       13s         1       {deployment-controller }                Normal
  13s       13s         1       {deployment-controller }                Normal
```

To fix this, we need to rollback to a previous revision of Deployment that is stable.

Checking Rollout History of a Deployment

First, check the revisions of this deployment:

```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl create -f docs/user-guide/nginx-deployment.yaml --record
2           kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
3           kubectl set image deployment/nginx-deployment nginx=nginx:1.91
```

Because we recorded the command while creating this Deployment using `--record`, we can easily see the changes we made in each revision.

To further see the details of each revision, run:

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Annotations:  kubernetes.io/change-cause=kubectl set image deployment/nginx-depl
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

Rolling Back to a Previous Revision

Now we've decided to undo the current rollout and rollback to the previous revision:

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
```

Alternatively, you can rollback to a specific revision by specify that in `--to-revision`:

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=2
deployment "nginx-deployment" rolled back
```

For more details about rollout related commands, read `kubectl rollout`.

The Deployment is now rolled back to a previous stable revision. As you can see, a `DeploymentRollback` event for rolling back to revision 2 is generated from Deployment controller.

```
$ kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           30m

$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                     SubobjectPath   Type
  --------- --------    -----   ----                     -------------   --------
  30m       30m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  29m       29m         1       {deployment-controller }                 Normal
  2m        2m          1       {deployment-controller }                 Normal
  2m        2m          1       {deployment-controller }                 Normal
  29m       2m          2       {deployment-controller }                 Normal
```

Scaling a Deployment

You can scale a Deployment by using the following command:

```
$ kubectl scale deployment nginx-deployment --replicas=10
deployment "nginx-deployment" scaled
```

Assuming horizontal pod autoscaling is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

```
$ kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80
deployment "nginx-deployment" autoscaled
```

Proportional scaling

RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in progress or paused), then the Deployment controller will balance the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called *proportional scaling*.

For example, you are running a Deployment with 10 replicas, maxSurge=3, and maxUnavailable=2.

```
$ kubectl get deploy
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment     10        10        10           10          50s
```

You update to a new image which happens to be unresolvable from inside the cluster.

```
$ kubectl set image deploy/nginx-deployment nginx=nginx:sometag
deployment "nginx-deployment" image updated
```

The image update starts a new rollout with ReplicaSet nginx-deployment-1989198191, but it's blocked due to the maxUnavailable requirement that we mentioned above.

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   5         5         0         9s
nginx-deployment-618515232    8         8         8         1m
```

Then a new scaling request for the Deployment comes along. The autoscaler increments the Deployment replicas to 15. The Deployment controller needs to decide where to add these new 5 replicas. If we weren't using proportional scaling, all 5 of them would be added in the new ReplicaSet. With proportional scaling, we spread the additional replicas across all ReplicaSets. Bigger proportions go to the ReplicaSets with the most replicas and lower proportions go to ReplicaSets with less replicas. Any leftovers are added to the ReplicaSet with the most replicas. ReplicaSets with zero replicas are not scaled up.

In our example above, 3 replicas will be added to the old ReplicaSet and 2 replicas will be added to the new ReplicaSet. The rollout process should eventually move all replicas to the new ReplicaSet, assuming the new replicas become healthy.

```
$ kubectl get deploy
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment     15        18        7            8           7m
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   7         7         0         7m
nginx-deployment-618515232    11        11        11        7m
```

Pausing and Resuming a Deployment

You can pause a Deployment before triggering one or more updates and then resume it. This will allow you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

For example, with a Deployment that was just created:

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         1m
```

Pause by running the following command:

```
$ kubectl rollout pause deployment/nginx-deployment
deployment "nginx-deployment" paused
```

Then update the image of the Deployment:

```
$ kubectl set image deploy/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

Notice that no new rollout started:

```
$ kubectl rollout history deploy/nginx-deployment
deployments "nginx"
REVISION  CHANGE-CAUSE
1   <none>

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         2m
```

You can make as many updates as you wish, for example, update the resources that will be used:

```
$ kubectl set resources deployment nginx-deployment -c=nginx --limits=cpu=200m,mem
deployment "nginx-deployment" resource requirements updated
```

The initial state of the Deployment prior to pausing it will continue its function, but new updates to the Deployment will not have any effect as long as the Deployment is paused.

Eventually, resume the Deployment and observe a new ReplicaSet coming up with all the new updates:

```
$ kubectl rollout resume deploy/nginx-deployment
deployment "nginx" resumed
$ kubectl get rs -w
NAME               DESIRED    CURRENT    READY      AGE
nginx-2142116321   2          2          2          2m
nginx-3926361531   2          2          0          6s
nginx-3926361531   2          2          1          18s
nginx-2142116321   1          2          2          2m
nginx-2142116321   1          2          2          2m
nginx-3926361531   3          2          1          18s
nginx-3926361531   3          2          1          18s
nginx-2142116321   1          1          1          2m
nginx-3926361531   3          3          1          18s
nginx-3926361531   3          3          2          19s
nginx-2142116321   0          1          1          2m
nginx-2142116321   0          1          1          2m
nginx-2142116321   0          0          0          2m
nginx-3926361531   3          3          3          20s
^C
$ kubectl get rs
NAME               DESIRED    CURRENT    READY      AGE
nginx-2142116321   0          0          0          2m
nginx-3926361531   3          3          3          28s
```

Note: You cannot rollback a paused Deployment until you resume it.

Deployment status

A Deployment enters various states during its lifecycle. It can be progressing while rolling out a new ReplicaSet, it can be complete, or it can fail to progress.

Progressing Deployment

Kubernetes marks a Deployment as *progressing* when one of the following tasks is performed:

- The Deployment creates a new ReplicaSet.
- The Deployment is scaling up its newest ReplicaSet.
- The Deployment is scaling down its older ReplicaSet(s).

- New Pods become ready or available (ready for at least MinReadySeconds).

You can monitor the progress for a Deployment by using `kubectl rollout status`.

Complete Deployment

Kubernetes marks a Deployment as *complete* when it has the following characteristics:

- All of the replicas associated with the Deployment have been updated to the latest version you've specified, meaning any updates you've requested have been completed.
- All of the replicas associated with the Deployment are available.
- No old replicas for the Deployment are running.

You can check if a Deployment has completed by using `kubectl rollout status`. If the rollout completed successfully, `kubectl rollout status` returns a zero exit code.

```
$ kubectl rollout status deploy/nginx-deployment
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx" successfully rolled out
$ echo $?
0
```

Failed Deployment

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: (`spec.progressDeadlineSeconds`). `spec.progressDeadlineSeconds` denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

The following `kubectl` command sets the spec with `progressDeadlineSeconds` to make the controller report lack of progress for a Deployment after 10 minutes:

```
$ kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds"
"nginx-deployment" patched
```

Once the deadline has been exceeded, the Deployment controller adds a DeploymentCondition with the following attributes to the Deployment's `status.conditions` :

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

See the Kubernetes API conventions for more information on status conditions.

Note: Kubernetes will take no action on a stalled Deployment other than to report a status condition with `Reason=ProgressDeadlineExceeded` . Higher level orchestrators can take advantage of it and act accordingly, for example, rollback the Deployment to its previous version.

Note: If you pause a Deployment, Kubernetes does not check progress against your specified deadline. You can safely pause a Deployment in the middle of a rollout and resume without triggering the condition for exceeding the deadline.

You may experience transient errors with your Deployments, either due to a low timeout that you have set or due to any other kind of error that can be treated as transient. For example, let's suppose you have insufficient quota. If you describe the Deployment you will notice the following section:

```
$ kubectl describe deployment nginx-deployment
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

If you run `kubectl get deployment nginx-deployment -o yaml`, the Deployement status might look like this:

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progressing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: ex
      object-counts, requested: pods=1, used: pods=3, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

Eventually, once the Deployment progress deadline is exceeded, Kubernetes updates the status and the reason for the Progressing condition:

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

You can address an issue of insufficient quota by scaling down your Deployment, by scaling down other controllers you may be running, or by increasing quota in your namespace. If you satisfy the quota conditions and the Deployment controller then completes the Deployment rollout, you'll see the Deployment's status update with a successful condition (`Status=True` and `Reason=NewReplicaSetAvailable`).

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

`Type=Available` with `Status=True` means that your Deployment has minimum availability. Minimum availability is dictated by the parameters specified in the deployment strategy. `Type=Progressing` with `Status=True` means that your Deployment is either in the middle of a rollout and it is progressing or that it has successfully completed its progress and the minimum required new replicas are available (see the Reason of the condition for the particulars - in our case `Reason=NewReplicaSetAvailable` means that the Deployment is complete).

You can check if a Deployment has failed to progress by using `kubectl rollout status` . `kubectl rollout status` returns a non-zero exit code if the Deployment has exceeded the progression deadline.

```
$ kubectl rollout status deploy/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
$ echo $?
1
```

Operating on a failed deployment

All actions that apply to a complete Deployment also apply to a failed Deployment. You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment pod template.

Clean up Policy

You can set `.spec.revisionHistoryLimit` field in a Deployment to specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, all revision history will be kept. In a future version, it will default to switch to 2.

> **Note:** Explicitly setting this field to 0, will result in cleaning up all the history of your Deployment thus that Deployment will not be able to roll back.

Use Cases

Canary Deployment

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in managing resources.

Writing a Deployment Spec

As with all other Kubernetes configs, a Deployment needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and using kubectl to manage resources documents.

A Deployment also needs a `.spec` section.

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See selector).

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

Replicas

`.spec.replicas` is an optional field that specifies the number of desired Pods. It defaults to 1.

Selector

`.spec.selector` is an optional field that specifies a label selector for the Pods targeted by this deployment.

`.spec.selector` must match `.spec.template.metadata.labels`, or it will be rejected by the API.

In API version `apps/v1beta2`, `.spec.selector` and `.metadata.labels` no longer default to `.spec.template.metadata.labels` if not set. So they must be set explicitly. Also note that `.spec.selector` is immutable after creation of the Deployment in `apps/v1beta2`.

A Deployment may terminate Pods whose labels match the selector if their template is different from `.spec.template` or if the total number of such Pods exceeds `.spec.replicas`. It brings up new Pods with `.spec.template` if the number of Pods is less than the desired number.

Note: You should not create other pods whose labels match this selector, either directly, by creating another Deployment, or by creating another controller such as a ReplicaSet or a ReplicationController. If you do so, the first Deployment thinks that it created these other pods. Kubernetes does not stop you from doing this.

If you have multiple controllers that have overlapping selectors, the controllers will fight with each other and won't behave correctly.

Strategy

`.spec.strategy` specifies the strategy used to replace old Pods by new ones. `.spec.strategy.type` can be "Recreate" or "RollingUpdate". "RollingUpdate" is the default value.

Recreate Deployment

All existing Pods are killed before new ones are created when `.spec.strategy.type==Recreate`.

Rolling Update Deployment

The Deployment updates Pods in a rolling update fashion when `.spec.strategy.type==RollingUpdate`. You can specify `maxUnavailable` and `maxSurge` to control the rolling update process.

Max Unavailable

`.spec.strategy.rollingUpdate.maxUnavailable` is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if `.spec.strategy.rollingUpdate.maxSurge` is 0. The default value is 25%.

For example, when this value is set to 30%, the old ReplicaSet can be scaled down to 70% of desired Pods immediately when the rolling update starts. Once new Pods are ready, old ReplicaSet can be scaled down further, followed by scaling up the new ReplicaSet, ensuring that the total number of Pods available at all times during the update is at least 70% of the desired Pods.

Max Surge

`.spec.strategy.rollingUpdate.maxSurge` is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if

`MaxUnavailable` is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

For example, when this value is set to 30%, the new ReplicaSet can be scaled up immediately when the rolling update starts, such that the total number of old and new Pods does not exceed 130% of desired Pods. Once old Pods have been killed, the new ReplicaSet can be scaled up further, ensuring that the total number of Pods running at any time during the update is at most 130% of desired Pods.

Progress Deadline Seconds

`.spec.progressDeadlineSeconds` is an optional field that specifies the number of seconds you want to wait for your Deployment to progress before the system reports back that the Deployment has failed progressing - surfaced as a condition with `Type=Progressing` , `Status=False` . and `Reason=ProgressDeadlineExceeded` in the status of the resource. The deployment controller will keep retrying the Deployment. In the future, once automatic rollback will be implemented, the deployment controller will roll back a Deployment as soon as it observes such a condition.

If specified, this field needs to be greater than `.spec.minReadySeconds` .

Min Ready Seconds

`.spec.minReadySeconds` is an optional field that specifies the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. This defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see Container Probes.

Rollback To

Field `.spec.rollbackTo` has been deprecated in API versions `extensions/v1beta1` and `apps/v1beta1` , and is no longer supported in API version `apps/v1beta2` . Instead, `kubectl rollout undo` as introduced in Rolling Back to a Previous Revision should be used.

Revision History Limit

A Deployment's revision history is stored in the replica sets it controls.

`.spec.revisionHistoryLimit` is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. Its ideal value depends on the frequency and stability of new Deployments. All old ReplicaSets will be kept by default, consuming resources in `etcd` and crowding the output of `kubectl get rs`, if this field is not set. The configuration of each Deployment revision is stored in its ReplicaSets; therefore, once an old ReplicaSet is deleted, you lose the ability to rollback to that revision of Deployment.

More specifically, setting this field to zero means that all old ReplicaSets with 0 replica will be cleaned up. In this case, a new Deployment rollout cannot be undone, since its revision history is cleaned up.

Paused

`.spec.paused` is an optional boolean field for pausing and resuming a Deployment. The only difference between a paused Deployment and one that is not paused, is that any changes into the PodTemplateSpec of the paused Deployment will not trigger new rollouts as long as it is paused. A Deployment is not paused by default when it is created.

Alternative to Deployments

kubectl rolling update

Kubectl rolling update updates Pods and ReplicationControllers in a similar fashion. But Deployments are recommended, since they are declarative, server side, and have additional features, such as rolling back to any previous revision even after the rolling update is done.

StatefulSets

StatefulSet is the workload API object used to manage stateful applications. StatefulSets are beta in 1.8.

Manage the deployment and scaling of a set of Pods, *and provide guarantees about ordering*. They do so by maintaining a *unique*, sticky identity for each of their Pods.

Like Deployments, StatefulSets manage Pods that are based on an identical container spec. However, although their specs are the same, the Pods in a StatefulSet are not interchangeable. Each Pod has a persistent identifier that it maintains across any rescheduling.

StatefulSets also operate according to the Controller pattern. You define your desired state in a StatefulSet *object*, and the StatefulSet *controller* makes any necessary updates to the get there from the current state.

Using StatefulSets

StatefulSets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, graceful deletion and termination.
- Ordered, automated rolling updates.

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn't require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application with a controller that provides a set of stateless replicas. Controllers such as Deployment or ReplicaSet may be better suited to your stateless needs.

Limitations

- StatefulSet is a beta resource, not available in any Kubernetes release prior to 1.5.
- As with all alpha/beta resources, you can disable StatefulSet through the `--runtime-config` option passed to the apiserver.
- The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested `storage class`, or pre-provisioned by an admin.
- Deleting and/or scaling a StatefulSet down will *not* delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.

Components

The example below demonstrates the components of a StatefulSet.

- A Headless Service, named nginx, is used to control the network domain.
- The StatefulSet, named web, has a Spec that indicates that 3 replicas of the nginx container will be launched in unique Pods.
- The volumeClaimTemplates will provide stable storage using PersistentVolumes provisioned by a PersistentVolume Provisioner.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: my-storage-class
      resources:
        requests:
          storage: 1Gi
```

Pod Selector

You must set the `spec.selector` field of a StatefulSet to match the labels of its `.spec.template.metadata.labels`. Prior to Kubernetes 1.8, the `spec.selector` field was defaulted when omitted. In 1.8 and later versions, failing to specify a matching Pod Selector will result in a validation error during StatefulSet creation.

Pod Identity

StatefulSet Pods have a unique identity that is comprised of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it's (re)scheduled on.

Ordinal Index

For a StatefulSet with N replicas, each Pod in the StatefulSet will be assigned an integer ordinal, in the range [0,N), that is unique over the Set.

Stable Network ID

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed hostname is `$(statefulset name)-$(ordinal)`. The example above will create three Pods named `web-0,web-1,web-2`. A StatefulSet can use a Headless Service to control the domain of its Pods. The domain managed by this Service takes the form: `$(service name).$(namespace).svc.cluster.local`, where "cluster.local" is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: `$(podname).$(governing service domain)`, where the governing service is defined by the `serviceName` field on the StatefulSet.

Here are some examples of choices for Cluster Domain, Service name, StatefulSet name, and how that affects the DNS names for the StatefulSet's Pods.

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	Pod DNS	Pod Hostname

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	Pod DNS	Pod Hostname
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.default.svc.cluster.local	web-{0..N-1}
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.svc.cluster.local	web-{0..N-1}
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.foo.svc.kube.local	web-{0..N-1}

Note that Cluster Domain will be set to `cluster.local` unless otherwise configured.

Stable Storage

Kubernetes creates one PersistentVolume for each VolumeClaimTemplate. In the nginx example above, each Pod will receive a single PersistentVolume with a StorageClass of `my-storage-class` and 1 Gib of provisioned storage. If no StorageClass is specified, then the default StorageClass will be used. When a Pod is (re)scheduled onto a node, its `volumeMounts` mount the PersistentVolumes associated with its PersistentVolume Claims. Note that, the PersistentVolumes associated with the Pods' PersistentVolume Claims are not deleted when the Pods, or StatefulSet are deleted. This must be done manually.

Deployment and Scaling Guarantees

- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
- When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
- Before a Pod is terminated, all of its successors must be completely shutdown.

The StatefulSet should not specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. This practice is unsafe and strongly discouraged. For further explanation, please refer to force deleting StatefulSet Pods.

When the nginx example above is created, three Pods will be deployed in the order web-0, web-1, web-2. web-1 will not be deployed before web-0 is Running and Ready, and web-2 will not be deployed until web-1 is Running and Ready. If web-0 should fail, after web-1 is Running and Ready, but before web-2 is launched, web-2 will not be launched until web-0 is successfully relaunched and becomes Running and Ready.

If a user were to scale the deployed example by patching the StatefulSet such that `replicas=1`, web-2 would be terminated first. web-1 would not be terminated until web-2 is fully shutdown and deleted. If web-0 were to fail after web-2 has been terminated and is completely shutdown, but prior to web-1's termination, web-1 would not be terminated until web-0 is Running and Ready.

Pod Management Policies

In Kubernetes 1.7 and later, StatefulSet allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its `.spec.podManagementPolicy` field.

OrderedReady Pod Management

`OrderedReady` pod management is the default for StatefulSets. It implements the behavior described above.

Parallel Pod Management

`Parallel` pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod.

Update Strategies

In Kubernetes 1.7 and later, StatefulSet's `.spec.updateStrategy` field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet.

On Delete

The `OnDelete` update strategy implements the legacy (1.6 and prior) behavior. It is the default strategy when `spec.updateStrategy` is left unspecified. When a StatefulSet's `.spec.updateStrategy.type` is set to `OnDelete`, the StatefulSet controller will not automatically update the Pods in a StatefulSet. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet's `.spec.template`.

Rolling Updates

The `RollingUpdate` update strategy implements automated, rolling update for the Pods in a StatefulSet. When a StatefulSet's `.spec.updateStrategy.type` is set to `RollingUpdate`, the StatefulSet controller will delete and recreate each Pod in the StatefulSet. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time. It will wait until an updated Pod is Running and Ready prior to updating its predecessor.

Partitions

The `RollingUpdate` update strategy can be partitioned, by specifying a `.spec.updateStrategy.rollingUpdate.partition`. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's `.spec.template` is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version. If a StatefulSet's `.spec.updateStrategy.rollingUpdate.partition` is greater than its `.spec.replicas`, updates to its `.spec.template` will not be propagated to its Pods. In most cases you will not need to use a partition, but they are useful if you want to stage an update, roll out a canary, or perform a phased roll out.

What's next

- Follow an example of [deploying a stateful application](#).
- Follow an example of [deploying Cassandra with Stateful Sets](#).

Daemon Sets

What is a DaemonSet?

A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon, such as `glusterd`, `ceph`, on each node.
- running a logs collection daemon on every node, such as `fluentd` or `logstash`.
- running a node monitoring daemon on every node, such as Prometheus Node Exporter, `collectd`, Datadog agent, New Relic agent, or Ganglia `gmond`.

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

Writing a DaemonSet Spec

Create a DaemonSet

You can describe a DaemonSet in a YAML file. For example, the daemonset.yaml file below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

daemonset.yaml

```yaml
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: gcr.io/google-containers/fluentd-elasticsearch:1.20
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

- Create a DaemonSet based on the YAML file: `kubectl create -f daemonset.yaml`

Required Fields

As with all other Kubernetes config, a DaemonSet needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and working with resources documents.

A DaemonSet also needs a `.spec` section.

Pod Template

The `.spec.template` is one of the required fields in `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a DaemonSet has to specify appropriate labels (see pod selector).

A Pod Template in a DaemonSet must have a `RestartPolicy` equal to `Always`, or be unspecified, which defaults to `Always`.

Pod Selector

The `.spec.selector` field is a pod selector. It works the same as the `.spec.selector` of a Job.

As of Kubernetes 1.8, you must specify a pod selector that matches the labels of the `.spec.template`. The pod selector will no longer be defaulted when left empty. Selector defaulting was not compatible with `kubectl apply`. Also, once a DaemonSet is created, its `spec.selector` can not be mutated. Mutating the pod selector can lead to the unintentional orphaning of Pods, and it was found to be confusing to users.

The `spec.selector` is an object consisting of two fields:

- `matchLabels` - works the same as the `.spec.selector` of a ReplicationController.
- `matchExpressions` - allows to build more sophisticated selectors by specifying key, list of values and an operator that relates the key and values.

When the two are specified the result is ANDed.

If the `.spec.selector` is specified, it must match the `.spec.template.metadata.labels`. If not specified, they are defaulted to be equal. Config with these not matching will be rejected by the API.

Also you should not normally create any Pods whose labels match this selector, either directly, via another DaemonSet, or via other controller such as ReplicaSet. Otherwise, the DaemonSet controller will think that those Pods were created by it. Kubernetes will not stop you from doing this. One case where you might want to do this is manually create a Pod with a different value on a node for testing.

If you attempt to create a DaemonSet such that

Running Pods on Only Some Nodes

If you specify a `.spec.template.spec.nodeSelector`, then the DaemonSet controller will create Pods on nodes which match that node selector. Likewise if you specify a `.spec.template.spec.affinity`, then DaemonSet controller will create Pods on nodes which match that node affinity. If you do not specify either, then the DaemonSet controller will create Pods on all nodes.

How Daemon Pods are Scheduled

Normally, the machine that a Pod runs on is selected by the Kubernetes scheduler. However, Pods created by the DaemonSet controller have the machine already selected (`.spec.nodeName` is specified when the Pod is created, so it is ignored by the scheduler). Therefore:

- The `unschedulable` field of a node is not respected by the DaemonSet controller.
- The DaemonSet controller can make Pods even when the scheduler has not been started, which can help cluster bootstrap.

Daemon Pods do respect taints and tolerations, but they are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.alpha.kubernetes.io/notReady`
- `node.alpha.kubernetes.io/unreachable`

This ensures that when the `TaintBasedEvictions` alpha feature is enabled, they will not be evicted when there are node problems such as a network partition. (When the `TaintBasedEvictions` feature is not enabled, they are also not evicted in these scenarios, but due to hard-coded behavior of the NodeController rather than due to tolerations).

They also tolerate following `NoSchedule` taints:

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`

When the support to critical pods is enabled and the pods in a DaemonSet are labelled as critical, the Daemon pods are created with an additional `NoSchedule` toleration for the `node.kubernetes.io/out-of-disk` taint.

Note that all above `NoSchedule` taints above are created only in version 1.8 or later if the alpha feature `TaintNodesByCondition` is enabled.

Communicating with Daemon Pods

Some possible patterns for communicating with Pods in a DaemonSet are:

- **Push**: Pods in the DaemonSet are configured to send updates to another service, such as a stats database. They do not have clients.
- **NodeIP and Known Port**: Pods in the DaemonSet can use a `hostPort`, so that the pods are reachable via the node IPs. Clients know the list of node IPs somehow, and know the port by convention.
- **DNS**: Create a headless service with the same pod selector, and then discover DaemonSets using the `endpoints` resource or retrieve multiple A records from DNS.
- **Service**: Create a service with the same Pod selector, and use the service to reach a daemon on a random node. (No way to reach specific node.)

Updating a DaemonSet

If node labels are changed, the DaemonSet will promptly add Pods to newly matching nodes and delete Pods from newly not-matching nodes.

You can modify the Pods that a DaemonSet creates. However, Pods do not allow all fields to be updated. Also, the DaemonSet controller will use the original template the next time a node (even with the same name) is created.

You can delete a DaemonSet. If you specify `--cascade=false` with `kubectl`, then the Pods will be left on the nodes. You can then create a new DaemonSet with a different template. The new DaemonSet with the different template will recognize all the existing Pods as having matching labels. It will not modify or delete them despite a mismatch in the Pod template. You will need to force new Pod creation by deleting the Pod or deleting the node.

In Kubernetes version 1.6 and later, you can perform a rolling update on a DaemonSet.

Future releases of Kubernetes will support controlled updating of nodes.

Alternatives to DaemonSet

Init Scripts

It is certainly possible to run daemon processes by directly starting them on a node (e.g. using `init`, `upstartd`, or `systemd`). This is perfectly fine. However, there are several advantages to running such processes via a DaemonSet:

- Ability to monitor and manage logs for daemons in the same way as applications.
- Same config language and tools (e.g. Pod templates, `kubectl`) for daemons and applications.
- Future versions of Kubernetes will likely support integration between DaemonSet-created Pods and node upgrade workflows.
- Running daemons in containers with resource limits increases isolation between daemons from app containers. However, this can also be accomplished by running the daemons in a container but not in a Pod (e.g. start directly via Docker).

Bare Pods

It is possible to create Pods directly which specify a particular node to run on. However, a DaemonSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, you should use a DaemonSet rather than creating individual Pods.

Static Pods

It is possible to create Pods by writing a file to a certain directory watched by Kubelet. These are called static pods. Unlike DaemonSet, static Pods cannot be managed with kubectl or other Kubernetes API clients. Static Pods do not depend on the apiserver, making them useful in cluster bootstrapping cases. Also, static Pods may be deprecated in the future.

Deployments

DaemonSets are similar to Deployments in that they both create Pods, and those Pods have processes which are not expected to terminate (e.g. web servers, storage servers).

Use a Deployment for stateless services, like frontends, where scaling up and down the number of replicas and rolling out updates are more important than controlling exactly which host the Pod runs on. Use a DaemonSet when it is important that a copy of a Pod always run on all or certain hosts, and when it needs to start before other Pods.

Garbage Collection

The role of the Kubernetes garbage collector is to delete certain objects that once had an owner, but no longer have an owner.

Note: Garbage collection is a beta feature and is enabled by default in Kubernetes version 1.4 and later.

Owners and dependents

Some Kubernetes objects are owners of other objects. For example, a ReplicaSet is the owner of a set of Pods. The owned objects are called *dependents* of the owner object. Every dependent object has a `metadata.ownerReferences` field that points to the owning object.

Sometimes, Kubernetes sets the value of `ownerReference` automatically. For example, when you create a ReplicaSet, Kubernetes automatically sets the `ownerReference` field of each Pod in the ReplicaSet. In 1.6, Kubernetes automatically sets the value of `ownerReference` for objects created or adopted by ReplicationController, ReplicaSet, StatefulSet, DaemonSet, and Deployment.

You can also specify relationships between owners and dependents by manually setting the `ownerReference` field.

Here's a configuration file for a ReplicaSet that has three Pods:

my-repset.yaml

my-repset.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: my-repset
spec:
  replicas: 3
  selector:
    matchLabels:
      pod-is-for: garbage-collection-example
  template:
    metadata:
      labels:
        pod-is-for: garbage-collection-example
    spec:
      containers:
      - name: nginx
        image: nginx
```

If you create the ReplicaSet and then view the Pod metadata, you can see OwnerReferences field:

```
kubectl create -f https://k8s.io/docs/concepts/abstractions/controllers/my-repset.
kubectl get pods --output=yaml
```

The output shows that the Pod owner is a ReplicaSet named my-repset:

```
apiVersion: v1
kind: Pod
metadata:
  ...
  ownerReferences:
  - apiVersion: extensions/v1beta1
    controller: true
    blockOwnerDeletion: true
    kind: ReplicaSet
    name: my-repset
    uid: d9607e19-f88f-11e6-a518-42010a800195
  ...
```

Controlling how the garbage collector deletes dependents

When you delete an object, you can specify whether the object's dependents are also deleted automatically. Deleting dependents automatically is called *cascading deletion*. There are two modes of *cascading deletion*: *background* and *foreground*.

If you delete an object without deleting its dependents automatically, the dependents are said to be *orphaned*.

Background cascading deletion

In *background cascading deletion*, Kubernetes deletes the owner object immediately and the garbage collector then deletes the dependents in the background.

Foreground cascading deletion

In *foreground cascading deletion*, the root object first enters a "deletion in progress" state. In the "deletion in progress" state, the following things are true:

- The object is still visible via the REST API
- The object's `deletionTimestamp` is set
- The object's `metadata.finalizers` contains the value "foregroundDeletion".

Once the "deletion in progress" state is set, the garbage collector deletes the object's dependents. Once the garbage collector has deleted all "blocking" dependents (objects with `ownerReference.blockOwnerDeletion=true`), it delete the owner object.

Note that in the "foregroundDeletion", only dependents with `ownerReference.blockOwnerDeletion` block the deletion of the owner object. Kubernetes version 1.7 will add an admission controller that controls user access to set `blockOwnerDeletion` to true based on delete permissions on the owner object, so that unauthorized dependents cannot delay deletion of an owner object.

If an object's `ownerReferences` field is set by a controller (such as Deployment or ReplicaSet), blockOwnerDeletion is set automatically and you do not need to manually modify this field.

Setting the cascading deletion policy

To control the cascading deletion policy, set the `deleteOptions.propagationPolicy` field on your owner object. Possible values include "Orphan", "Foreground", or "Background".

The default garbage collection policy for many controller resources is `orphan`, including ReplicationController, ReplicaSet, StatefulSet, DaemonSet, and Deployment. So unless you specify otherwise, dependent objects are orphaned.

Here's an example that deletes dependents in background:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces/default/replicase
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' \
-H "Content-Type: application/json"
```

Here's an example that deletes dependents in foreground:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces/default/replicase
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' \
-H "Content-Type: application/json"
```

Here's an example that orphans dependents:

```
kubectl proxy --port=8080
curl -X DELETE localhost:8080/apis/extensions/v1beta1/namespaces/default/replicase
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' \
-H "Content-Type: application/json"
```

kubectl also supports cascading deletion. To delete dependents automatically using kubectl, set `--cascade` to true. To orphan dependents, set `--cascade` to false. The default value for `--cascade` is true.

Here's an example that orphans the dependents of a ReplicaSet:

```
kubectl delete replicaset my-repset --cascade=false
```

Additional note on Deployments

When using cascading deletes with Deployments you *must* use `propagationPolicy: Foreground` to delete not only the ReplicaSets created, but also their Pods. If this type of *propagationPolicy* is not used, only the ReplicaSets will be deleted, and the Pods will be orphaned. See kubeadm/#149 for more information.

Known issues

Tracked at #26120

What's next

Design Doc 1

Design Doc 2

Jobs - Run to Completion

What is a Job?

A *job* creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the *job* tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. Deleting a Job will cleanup the pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first pod fails or is deleted (for example due to a node hardware failure or a node reboot).

A Job can also be used to run multiple pods in parallel.

Running an example Job

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
      backoffLimit: 4
```

Run the example job by downloading the example file and then running this command:

```
$ kubectl create -f ./job.yaml
job "pi" created
```

Check on the status of the job using this command:

```
$ kubectl describe jobs/pi
Name:             pi
Namespace:        default
Selector:         controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
Labels:           controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
                  job-name=pi
Annotations:      <none>
Parallelism:      1
Completions:      1
Start Time:       Tue, 07 Jun 2016 10:56:16 +0200
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=b1db589a-2c8d-11e6-b324-0209dc45a495
                job-name=pi
  Containers:
   pi:
    Image:      perl
    Port:
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen     LastSeen     Count     From              SubobjectPath     Type        R
  ---------     --------     -----     ----              -------------     --------    -
  1m            1m           1         {job-controller }                   Normal      S
```

To view completed pods of a job, use `kubectl get pods --show-all`. The `--show-all` will show completed pods too.

To list all the pods that belong to a job in a machine readable form, you can use a command like this:

```
$ pods=$(kubectl get pods  --show-all --selector=job-name=pi --output=jsonpath={.i
echo $pods
pi-aiw0a
```

Here, the selector is the same as the selector for the job. The `--output=jsonpath` option specifies an expression that just gets the name from each pod in the returned list.

View the standard output of one of the pods:

```
$ kubectl logs $pods
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089
```

Writing a Job Spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `metadata` fields.

A Job also needs a `.spec` section.

Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a job must specify appropriate labels (see pod selector) and an appropriate restart policy.

Only a `RestartPolicy` equal to `Never` or `OnFailure` is allowed.

Pod Selector

The `.spec.selector` field is optional. In almost all cases you should not specify it. See section specifying your own pod selector.

Parallel Jobs

There are three main types of jobs:

1. Non-parallel Jobs
    1. normally only one pod is started, unless the pod fails.
    2. job is complete as soon as Pod terminates successfully.
2. Parallel Jobs with a *fixed completion count*:

1. specify a non-zero positive value for `.spec.completions`.
2. the job is complete when there is one successful pod for each value in the range 1 to `.spec.completions`.
3. **not implemented yet:** each pod passed a different index in the range 1 to `.spec.completions`.

3. Parallel Jobs with a *work queue*: - do not specify `.spec.completions`, default to `.spec.Parallelism`. - the pods must coordinate with themselves or an external service to determine what each should work on.

   1. each pod is independently capable of determining whether or not all its peers are done, thus the entire Job is done.
   2. when *any* pod terminates with success, no new pods are created.
   3. once at least one pod has terminated with success and all pods are terminated, then the job is completed with success.
   4. once any pod has exited with success, no other pod should still be doing any work or writing any output. They should all be in the process of exiting.

For a Non-parallel job, you can leave both `.spec.completions` and `.spec.parallelism` unset. When both are unset, both are defaulted to 1.

For a Fixed Completion Count job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1.

For a Work Queue Job, you must leave `.spec.completions` unset, and set `.spec.parallelism` to a non-negative integer.

For more information about how to make use of the different types of job, see the job patterns section.

Controlling Parallelism

The requested parallelism (`.spec.parallelism`) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

A job can be scaled up using the `kubectl scale` command. For example, the following command sets `.spec.parallelism` of a job called `myjob` to 10:

```
$ kubectl scale  --replicas=$N jobs/myjob
job "myjob" scaled
```

You can also use the `scale` subresource of the Job resource.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety or reasons:

- For Fixed Completion Count jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of `.spec.parallelism` are effectively ignored.
- For work queue jobs, no new pods are started after any pod has succeeded – remaining pods are allowed to complete, however.
- If the controller has not had time to react.
- If the controller failed to create pods for any reason (lack of ResourceQuota, lack of permission, etc.), then there may be fewer pods than requested.
- The controller may throttle new pod creation due to excessive previous pod failures in the same Job.
- When a pod is gracefully shutdown, it takes time to stop.

Handling Pod and Container Failures

A Container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the Container was killed for exceeding a memory limit, etc. If this happens, and the `.spec.template.spec.restartPolicy = "OnFailure"`, then the Pod stays on the node, but the Container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify `.spec.template.spec.restartPolicy = "Never"`. See pods-states for more information on `restartPolicy`.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the `.spec.template.spec.restartPolicy = "Never"`. When a Pod fails, then the Job controller starts a new Pod. Therefore, your program needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

Note that even if you specify `.spec.parallelism = 1` and `.spec.completions = 1` and `.spec.template.spec.restartPolicy = "Never"`, the same program may sometimes be started twice.

If you do specify `.spec.parallelism` and `.spec.completions` both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

Pod Backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so set `.spec.template.spec.backoffLimit` to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s …) capped at six minutes, The back-off limit is reset if no new failed Pods appear before the Job's next status check.

Job Termination and Cleanup

When a Job completes, no more Pods are created, but the Pods are not deleted either. Since they are terminated, they don't show up with `kubectl get pods`, but they will show up with `kubectl get pods -a`. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with `kubectl` (e.g. `kubectl delete jobs/pi` or `kubectl delete -f ./job.yaml`). When you delete the job using `kubectl`, all the pods it created are deleted too.

If a Job's pods are failing repeatedly, the Job will keep creating new pods forever, by default. Retrying forever can be a useful pattern. If an external dependency of the Job's pods is missing (for example an input file on a networked storage volume is not present), then the Job will keep trying Pods, and when you later resolve the external dependency (for example, creating the missing file) the Job will then complete without any further action.

However, if you prefer not to retry forever, you can set a deadline on the job. Do this by setting the `spec.activeDeadlineSeconds` field of the job to a number of seconds. The job will have status with `reason: DeadlineExceeded`. No more pods will be created, and existing pods will be deleted.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-with-timeout
spec:
  activeDeadlineSeconds: 100
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
      backoffLimit: 5
```

Note that both the Job Spec and the Pod Template Spec within the Job have a field with the same name. Set the one on the Job.

Job Patterns

The Job object can be used to support reliable parallel execution of Pods. The Job object is not designed to support closely-communicating parallel processes, as commonly found in scientific computing. It does support parallel processing of a set of independent but related *work items*. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a *batch job*.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

- One Job object for each work item, vs. a single Job object for all work items. The latter is better for large numbers of work items. The former creates some overhead for the user and for the system to manage large numbers of Job objects. Also, with the latter, the resource usage of the job (number of concurrently running pods) can be easily adjusted using the `kubectl scale` command.
- Number of pods created equals number of work items, vs. each pod can process multiple work items. The former typically requires less modification to existing code and containers. The latter is better for large numbers of work items, for similar reasons to the previous bullet.
- Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

Pattern	Single Job object	Fewer pods than work items?	Use app unmodified?	Works in Kube 1.1?
Job Template Expansion			✓	✓
Queue with Pod Per Work Item	✓		sometimes	✓
Queue with Variable Pod Count	✓	✓		✓
Single Job with Static Work Assignment	✓		✓	

When you specify completions with `.spec.completions`, each Pod created by the Job controller has an identical `spec`. This means that all pods will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for `.spec.parallelism` and `.spec.completions` for each of the patterns. Here, `W` is the number of work items.

Pattern	`.spec.completions`	`.spec.parallelism`
Job Template Expansion	1	should be 1
Queue with Pod Per Work Item	W	any
Queue with Variable Pod Count	1	any
Single Job with Static Work Assignment	W	any

Advanced Usage

Specifying your own pod selector

Normally, when you create a job object, you do not specify `spec.selector`. The system defaulting logic adds this field when the job is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the `spec.selector` of the job.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that job, and which matches unrelated pods, then pods of the unrelated job may be deleted, or this job may count other pods as completing it, or one or both of the jobs may refuse to create pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. ReplicationController) and their pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying `spec.selector`.

Here is an example of a case when you might want to use this feature.

Say job `old` is already running. You want existing pods to keep running, but you want the rest of the pods it creates to use a different pod template and for the job to have a new name. You cannot update the job because these fields are not updatable. Therefore, you delete job `old` but leave its pods running, using `kubectl delete jobs/old --cascade=false`. Before deleting it, you make a note of what selector it uses:

```
kind: Job
metadata:
  name: old
  ...
spec:
  selector:
    matchLabels:
      job-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

Then you create a new job with name `new` and you explicitly specify the same selector. Since the existing pods have label `job-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002`, they are controlled by job `new` as well.

You need to specify `manualSelector: true` in the new job since you are not using the selector that the system normally generates for you automatically.

```
kind: Job
metadata:
  name: new
  ...
spec:
  manualSelector: true
  selector:
    matchLabels:
      job-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
  ...
```

The new Job itself will have a different uid from `a8f3d00d-c6d2-11e5-9f87-42010af00002`. Setting `manualSelector: true` tells the system to that you know what you are doing and to allow this mismatch.

Alternatives

Bare Pods

When the node that a pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new pods to replace terminated ones. For this reason, we recommend that you use a job rather than a bare pod, even if your application requires only a single pod.

Replication Controller

Jobs are complementary to Replication Controllers. A Replication Controller manages pods which are not expected to terminate (e.g. web servers), and a Job manages pods that are expected to terminate (e.g. batch jobs).

As discussed in Pod Lifecycle, `Job` is *only* appropriate for pods with `RestartPolicy` equal to `OnFailure` or `Never`. (Note: If `RestartPolicy` is not set, the default value is `Always`.)

Single Job starts Controller Pod

Another pattern is for a single Job to create a pod which then creates other pods, acting as a sort of custom controller for those pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

One example of this pattern would be a Job which starts a Pod which runs a script that in turn starts a Spark master controller (see spark example), runs a spark driver, and then cleans up.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but complete control over what pods are created and how work is assigned to them.

Cron Jobs

Support for creating Jobs at specified times/dates (i.e. cron) is available in Kubernetes 1.4. More information is available in the cron job documents

Cron Jobs

What is a cron job?

A *Cron Job* manages time based Jobs, namely:

- Once at a specified point in time
- Repeatedly at a specified point in time

One CronJob object is like one line of a *crontab* (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

Note: The question mark (`?`) in the schedule has the same meaning as an asterisk `*`, that is, it stands for any of available value for a given field.

Note: CronJob resource in `batch/v2alpha1` API group has been deprecated starting from cluster version 1.8. You should switch to using `batch/v1beta1`, instead, which is enabled by default in the API server. Further in this document, we will be using `batch/v1beta1` in all the examples.

A typical use case is:

- Schedule a job execution at a given point in time.

- Create a periodic job, e.g. database backup, sending emails.

Prerequisites

You need a working Kubernetes cluster at version >= 1.8 (for CronJob). For previous versions of cluster (< 1.8) you need to explicitly enable `batch/v2alpha1` API by passing `--runtime-config=batch/v2alpha1=true` to the API server (see Turn on or off an API version for your cluster for more).

Creating a Cron Job

Here is an example Cron Job. Every minute, it runs a simple job to print current time and then say hello.

cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Run the example cron job by downloading the example file and then running this command:

```
$ kubectl create -f ./cronjob.yaml
cronjob "hello" created
```

Alternatively, use `kubectl run` to create a cron job without writing full config:

```
$ kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox -
cronjob "hello" created
```

After creating the cron job, get its status using this command:

```
$ kubectl get cronjob hello
NAME      SCHEDULE      SUSPEND   ACTIVE    LAST-SCHEDULE
hello     */1 * * * *   False     0         <none>
```

As you can see above, there's no active job yet, and no job has been scheduled, either.

Watch for the job to be created in around one minute:

```
$ kubectl get jobs --watch
NAME               DESIRED   SUCCESSFUL   AGE
hello-4111706356   1         1         2s
```

Now you've seen one running job scheduled by "hello". We can stop watching it and get the cron job again:

```
$ kubectl get cronjob hello
NAME      SCHEDULE      SUSPEND   ACTIVE    LAST-SCHEDULE
hello     */1 * * * *   False     0         Mon, 29 Aug 2016 14:34:00 -0700
```

You should see that "hello" successfully scheduled a job at the time specified in `LAST-SCHEDULE`. There are currently 0 active jobs, meaning that the job that's scheduled is completed or failed.

Now, find the pods created by the job last scheduled and view the standard output of one of the pods. Note that your job name and pod name would be different.

```
# Replace "hello-4111706356" with the job name in your system
$ pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=jsonpath={

$ echo $pods
hello-4111706356-o9qcm

$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
Hello from the Kubernetes cluster
```

Deleting a Cron Job

Once you don't need a cron job anymore, simply delete it with `kubectl`:

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

This stops new jobs from being created and removes all the jobs and pods created by this cronjob. You can read more about it in garbage collection section.

Cron Job Limitations

A cron job creates a job object *about* once per execution time of its schedule. We say "about" because there are certain circumstances where two jobs might be created, or no job might be created. We attempt to make these rare, but do not completely prevent them. Therefore, jobs should be *idempotent*.

The job is responsible for retrying pods, parallelism among pods it creates, and determining the success or failure of the set of pods. A cron job does not examine pods at all.

Writing a Cron Job Spec

As with all other Kubernetes configs, a cron job needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and using kubectl to manage resources documents.

A cron job also needs a `.spec` section.

Note: All modifications to a cron job, especially its `.spec`, will be applied only to the next run.

Schedule

The `.spec.schedule` is a required field of the `.spec`. It takes a Cron format string, e.g. `0 * * * *` or `@hourly`, as schedule time of its jobs to be created and executed.

Job Template

The `.spec.jobTemplate` is another required field of the `.spec`. It is a job template. It has exactly the same schema as a Job, except it is nested and does not have an `apiVersion` or `kind`, see Writing a Job Spec.

Starting Deadline Seconds

The `.spec.startingDeadlineSeconds` field is optional. It stands for the deadline (in seconds) for starting the job if it misses its scheduled time for any reason. Missed jobs executions will be counted as failed ones. If not specified, there's no deadline.

Concurrency Policy

The `.spec.concurrencyPolicy` field is also optional. It specifies how to treat concurrent executions of a job created by this cron job. Only one of the following concurrent policies may be specified:

- `Allow` (default): allows concurrently running jobs
- `Forbid`: forbids concurrent runs, skipping next run if previous hasn't finished yet
- `Replace`: cancels currently running job and replaces it with a new one

Note that concurrency policy only applies to the jobs created by the same cron job. If there are multiple cron jobs, their respective jobs are always allowed to run concurrently.

Suspend

The `.spec.suspend` field is also optional. If set to `true`, all subsequent executions will be suspended. It does not apply to already started executions. Defaults to false.

Jobs History Limits

The `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` fields are optional. These fields specify how many completed and failed jobs should be kept. By default, they are set to 3 and 1 respectively. Setting a limit to `0` corresponds to keeping none of the corresponding kind of jobs after they finish.

Configuration Best Practices

This document highlights and consolidates configuration best practices that are introduced throughout the user-guide, getting-started documentation, and examples.

This is a living document. If you think of something that is not on this list but might be useful to others, please don't hesitate to file an issue or submit a PR.

General Config Tips

- When defining configurations, specify the latest stable API version (currently v1).
- Configuration files should be stored in version control before being pushed to the cluster. This allows quick roll-back of a configuration if needed. It also aids with cluster re-creation and restoration if necessary.
- Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.
- Group related objects into a single file whenever it makes sense. One file is often easier to manage than several. See the guestbook-all-in-one.yaml file as an example of this syntax.

  Note also that many `kubectl` commands can be called on a directory, so you can also call `kubectl create` on a directory of config files. See below for more details.
- Don't specify default values unnecessarily, in order to simplify and minimize configs, and to reduce error. For example, omit the selector and labels in a `ReplicationController` if you want them to be the same as the labels in its `podTemplate`, since those fields are populated

from the `podTemplate` labels by default. See the guestbook app's .yaml files for some examples of this.

- Put an object description in an annotation to allow better introspection.

“Naked” Pods vs Replication Controllers and Jobs

- If there is a viable alternative to naked pods (in other words: pods not bound to a replication controller), go with the alternative. Naked pods will not be rescheduled in the event of node failure.

  Replication controllers are almost always preferable to creating pods, except for some explicit `restartPolicy: Never` scenarios. A Job object (currently in Beta) may also be appropriate.

Services

- It's typically best to create a service before corresponding replication controllers. This lets the scheduler spread the pods that comprise the service.

  You can also use this process to ensure that at least one replica works before creating lots of them:

  1. Create a replication controller without specifying replicas (this will set replicas=1);
  2. Create a service;
  3. Then scale up the replication controller.

- Don't use `hostPort` unless it is absolutely necessary (for example: for a node daemon). It specifies the port number to expose on the host. When you bind a Pod to a `hostPort`, there are a limited number of places to schedule a pod due to port conflicts— you can only schedule as many such Pods as there are nodes in your Kubernetes cluster.

  If you only need access to the port for debugging purposes, you can use the kubectl proxy and apiserver proxy or kubectl port-forward. You can use a Service object for external service access.

If you explicitly need to expose a pod's port on the host machine, consider using a NodePort service before resorting to `hostPort`.

- Avoid using `hostNetwork`, for the same reasons as `hostPort`.
- Use *headless services* for easy service discovery when you don't need kube-proxy load balancing. See headless services.

Using Labels

- Define and use labels that identify **semantic attributes** of your application or deployment. For example, instead of attaching a label to a set of pods to explicitly represent some service (For example, `service: myservice`), or explicitly representing the replication controller managing the pods (for example, `controller: mycontroller`), attach labels that identify semantic attributes, such as **{ app: myapp, tier: frontend, phase: test, deployment: v3 }**. This will let you select the object groups appropriate to the context— for example, a service for all "tier: frontend" pods, or all "test" phase components of app "myapp". See the guestbook app for an example of this approach.

  A service can be made to span multiple deployments, such as is done across rolling updates, by simply omitting release-specific labels from its selector, rather than updating a service's selector to match the replication controller's selector fully.

- To facilitate rolling updates, include version info in replication controller names, for example as a suffix to the name. It is useful to set a 'version' label as well. The rolling update creates a new controller as opposed to modifying the existing controller. So, there will be issues with version-agnostic controller names. See the documentation on the rolling-update command for more detail.

  Note that the Deployment object obviates the need to manage replication controller 'version names'. A desired state of an object is described by a Deployment, and if changes to that spec are *applied*, the deployment controller changes the actual state to the desired state at a controlled rate. (Deployment objects are currently part of the `extensions` API Group.)

- You can manipulate labels for debugging. Because Kubernetes replication controllers and services match to pods using labels, this allows you to remove a pod from being considered by a controller, or served traffic by a service, by removing the relevant selector labels. If you remove the labels of an existing pod, its controller will create a new pod to take its place. This is a useful way to debug a previously "live" pod in a quarantine environment. See the `kubectl label` command.

Container Images

- The default container image pull policy is `IfNotPresent`, which causes the Kubelet to not pull an image if it already exists. If you would like to always force a pull, you must specify a pull image policy of `Always` in your .yaml file (`imagePullPolicy: Always`) or specify a `:latest` tag on your image.

  That is, if you're specifying an image with other than the `:latest` tag, for example `myimage:v1`, and there is an image update to that same tag, the Kubelet won't pull the updated image. You can address this by ensuring that any updates to an image bump the image tag as well (for example, `myimage:v2`), and ensuring that your configs point to the correct version.

  Note: You should avoid using `:latest` tag when deploying containers in production, because this makes it hard to track which version of the image is running and hard to roll back.

- To work only with a specific version of an image, you can specify an image with its digest (SHA256). This approach guarantees that the image will never update. For detailed information about working with image digests, see the Docker documentation.

Using kubectl

- Use `kubectl create -f <directory>` where possible. This looks for config objects in all `.yaml`, `.yml`, and `.json` files in `<directory>` and passes them to `create`.
- Use `kubectl delete` rather than `stop`. `Delete` has a superset of the functionality of `stop`, and `stop` is deprecated.
- Use kubectl bulk operations (via files and/or labels) for get and delete. See label selectors and using labels effectively.

- Use `kubectl run` and `expose` to quickly create and expose single container Deployments. See the quick start guide for an example.

Managing Compute Resources for Containers

When you specify a Pod, you can optionally specify how much CPU and memory (RAM) each Container needs. When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner. For more details about the difference between requests and limits, see Resource QoS.

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes.

CPU and memory are collectively referred to as *compute resources*, or just *resources*. Compute resources are measurable quantities that can be requested, allocated, and consumed. They are distinct from API resources. API resources, such as Pods and Services are objects that can be read and modified through the Kubernetes API server.

Resource requests and limits of Pod and Container

Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`

Although requests and limits can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A *Pod resource request/limit* for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One cpu, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 *Hyperthread* on a bare-metal Intel processor with Hyperthreading

Fractional requests are allowed. A Container with `spec.containers[].resources.requests.cpu` of `0.5` is guaranteed half as much CPU as one that asks for 1 CPU. The expression `0.1` is equivalent to the expression `100m`, which can be read as “one hundred millicpu”. Some people say "one hundred millicores", and this is understood to mean the same thing. A request with a decimal point, like `0.1`, is converted to `100m` by the API, and precision finer than `1m` is not allowed. For this reason, the form `100m` might be preferred.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

Meaning of memory

Limits and requests for `memory` are measured in bytes. You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

```
128974848, 129e6, 129M, 123Mi
```

Here's an example. The following Pod has two Containers. Each Container has a request of 0.25 cpu and 64MiB (2^{26} bytes) of memory. Each Container has a limit of 0.5 cpu and 128MiB of memory. You can say the Pod has a request of 0.5 cpu and 128 MiB of memory, and a limit of 1 cpu and 256MiB of memory.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

How Pods with resource requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled Containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

How Pods with resource limits are run

When the kubelet starts a Container of a Pod, it passes the CPU and memory limits to the container runtime.

When using Docker:

- The `spec.containers[].resources.requests.cpu` is converted to its core value, which is potentially fractional, and multiplied by 1024. The greater of this number or 2 is used as the value of the `--cpu-shares` flag in the `docker run` command.
- The `spec.containers[].resources.limits.cpu` is converted to its millicore value and multiplied by 100. The resulting value is the total amount of CPU time that a container can use every 100ms. A container cannot use more than its share of CPU time during this interval.

  > **Note**: The default quota period is 100ms. The minimum resolution of CPU quota is 1ms.

- The `spec.containers[].resources.limits.memory` is converted to an integer, and used as the value of the `--memory` flag in the `docker run` command.

If a Container exceeds its memory limit, it might be terminated. If it is restartable, the kubelet will restart it, as with any other type of runtime failure.

If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory.

A Container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage.

To determine whether a Container cannot be scheduled or is being killed due to resource limits, see the Troubleshooting section.

Monitoring compute resource usage

The resource usage of a Pod is reported as part of the Pod status.

If optional monitoring is configured for your cluster, then Pod resource usage can be retrieved from the monitoring system.

Troubleshooting

My Pods are pending with event message failedScheduling

If the scheduler cannot find any node where a Pod can fit, the Pod remains unscheduled until a place can be found. An event is produced each time the scheduler fails to find a place for the Pod, like this:

```
$ kubectl describe pod frontend | grep -A 3 Events
Events:
  FirstSeen LastSeen   Count  From          Subobject   PathReason      Message
  36s   5s     6      {scheduler }              FailedScheduling  Failed for reaso
```

In the preceding example, the Pod named “frontend” fails to be scheduled due to insufficient CPU resource on the node. Similar error messages can also suggest failure due to insufficient memory (PodExceedsFreeMemory). In general, if a Pod is pending with a message of this type, there are several things to try:

- Add more nodes to the cluster.
- Terminate unneeded Pods to make room for pending Pods.
- Check that the Pod is not larger than all the nodes. For example, if all the nodes have a capacity of `cpu: 1`, then a Pod with a request of `cpu: 1.1` will never be scheduled.

You can check node capacities and amounts allocated with the `kubectl describe nodes` command. For example:

```
$ kubectl describe nodes e2e-test-minion-group-4lw4
Name:            e2e-test-minion-group-4lw4
[ ... lines removed for clarity ...]
Capacity:
 alpha.kubernetes.io/nvidia-gpu:    0
 cpu:                               2
 memory:                            7679792Ki
 pods:                              110
Allocatable:
 alpha.kubernetes.io/nvidia-gpu:    0
 cpu:                               1800m
 memory:                            7474992Ki
 pods:                              110
[ ... lines removed for clarity ...]
Non-terminated Pods:        (5 in total)
  Namespace    Name                                  CPU Requests  CPU Limits  Mem
  ---------    ----                                  ------------  ----------  ---
  kube-system  fluentd-gcp-v1.38-28bv1               100m (5%)     0 (0%)      200
  kube-system  kube-dns-3297075139-61lj3             260m (13%)    0 (0%)      100
  kube-system  kube-proxy-e2e-test-...               100m (5%)     0 (0%)      0 (
  kube-system  monitoring-influxdb-grafana-v4-z1m12  200m (10%)    200m (10%)  600
  kube-system  node-problem-detector-v0.1-fj7m3      20m (1%)      200m (10%)  20M
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests    CPU Limits    Memory Requests    Memory Limits
  ------------    ----------    ---------------    -------------
  680m (34%)      400m (20%)    920Mi (12%)        1070Mi (14%)
```

In the preceding output, you can see that if a Pod requests more than 1120m CPUs or 6.23Gi of memory, it will not fit on the node.

By looking at the `Pods` section, you can see which Pods are taking up space on the node.

The amount of resources available to Pods is less than the node capacity, because system daemons use a portion of the available resources. The `allocatable` field NodeStatus gives the amount of resources that are available to Pods. For more information, see Node Allocatable Resources.

The resource quota feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

My Container is terminated

Your Container might get terminated because it is resource-starved. To check whether a Container is being killed because it is hitting a resource limit, call `kubectl describe pod` on the Pod of interest:

```
[12:54:41] $ kubectl describe pod simmemleak-hra99
Name:                           simmemleak-hra99
Namespace:                      default
Image(s):                       saadali/simmemleak
Node:                           kubernetes-node-tf0f/10.240.216.66
Labels:                         name=simmemleak
Status:                         Running
Reason:
Message:
IP:                             10.244.2.75
Replication Controllers:        simmemleak (1/1 replicas created)
Containers:
  simmemleak:
    Image:  saadali/simmemleak
    Limits:
      cpu:                      100m
      memory:                   50Mi
    State:                      Running
      Started:                  Tue, 07 Jul 2015 12:54:41 -0700
    Last Termination State:     Terminated
      Exit Code:                1
      Started:                  Fri, 07 Jul 2015 12:54:30 -0700
      Finished:                 Fri, 07 Jul 2015 12:54:33 -0700
    Ready:                      False
    Restart Count:              5
Conditions:
  Type      Status
  Ready     False
Events:
  FirstSeen                         LastSeen                         Count  From
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {sched
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubel
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubel
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubel
  Tue, 07 Jul 2015 12:53:51 -0700   Tue, 07 Jul 2015 12:53:51 -0700  1      {kubel
```

In the preceding example, the `Restart Count: 5` indicates that the `simmemleak` Container in the Pod was terminated and restarted five times.

You can call `kubectl get pod` with the `-o go-template=...` option to fetch the status of previously terminated Containers:

```
[13:59:01] $ kubectl get pod -o go-template='{{range.status.containerStatuses}}{{"
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM Killed startedAt:2015-07-07T
```

You can see that the Container was terminated because of `reason:OOM Killed`, where `OOM` stands for Out Of Memory.

Local ephemeral storage (alpha feature)

Kubernetes version 1.8 introduces a new resource, *ephemeral-storage* for managing local ephemeral storage. In each Kubernetes node, kubelet's root directory (/var/lib/kubelet by default) and log directory (/var/log) are stored on the root partition of the node. This partition is also shared and consumed by pods via EmptyDir volumes, container logs, image layers and container writable layers.

This partition is "ephemeral" and applications cannot expect any performance SLAs (Disk IOPS for example) from this partition. Local ephemeral storage management only applies for the root partition; the optional partition for image layer and writable layer is out of scope.

> **Note:** If an optional runntime partition is used, root parition will not hold any image layer or writable layers.

Requests and limits setting for local ephemeral storage

Each Container of a Pod can specify one or more of the following:

- `spec.containers[].resources.limits.ephemeral-storage`
- `spec.containers[].resources.requests.ephemeral-storage`

Limits and requests for `ephemeral-storage` are measured in bytes. You can express storage as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

```
128974848, 129e6, 129M, 123Mi
```

For example, the following Pod has two Containers. Each Container has a request of 2GiB of local ephemeral storage. Each Container has a limit of 4GiB of local ephemeral storage. Therefore, the Pod has a request of 4GiB of local ephemeral storage, and a limit of 8GiB of storage.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
  - name: wp
    image: wordpress
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
```

How Pods with ephemeral-storage requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum amount of local ephemeral storage it can provide for Pods. (For more information, see “Node Allocatable” The scheduler ensures that the sum of the resource requests of the scheduled Containers is less than the capacity of the node.

How Pods with ephemeral-storage limits run

For container-level isolation, if a Container’s writable layer and logs usage exceeds its storage limit, the pod will be evicted. For pod-level isolation, if the sum of the local ephemeral storage usage from all containers and also the pod’s EmptyDir volumes exceeds the limit, the pod will be evicted.

Opaque integer resources (alpha feature)

DEPRECATION NOTICE: As of `Kubernetes v1.8`, this has been deprecated

Kubernetes version 1.5 introduces Opaque integer resources. Opaque integer resources allow cluster operators to advertise new node-level resources that would be otherwise unknown to the system.

Users can consume these resources in Pod specs just like CPU and memory. The scheduler takes care of the resource accounting so that no more than the available amount is simultaneously allocated to Pods.

> **Note:** Opaque Integer Resources will be removed in version 1.9. Extended Resources are a replacement for Opaque Integer Resources. Users can use any domain name prefix outside of the `kubernetes.io/` domain instead of the previous `pod.alpha.kubernetes.io/opaque-int-resource-` prefix.

Opaque integer resources are resources that begin with the prefix `pod.alpha.kubernetes.io/opaque-int-resource-`. The API server restricts quantities of these resources to whole numbers. Examples of *valid* quantities are `3`, `3000m` and `3Ki`. Examples of *invalid* quantities are `0.5` and `1500m`.

There are two steps required to use opaque integer resources. First, the cluster operator must advertise a per-node opaque resource on one or more nodes. Second, users must request the opaque resource in Pods.

To advertise a new opaque integer resource, the cluster operator should submit a `PATCH` HTTP request to the API server to specify the available quantity in the `status.capacity` for a node in the cluster. After this operation, the node's `status.capacity` will include a new resource. The `status.allocatable` field is updated automatically with the new resource asynchronously by the kubelet. Note that because the scheduler uses the node `status.allocatable` value when evaluating Pod fitness, there may be a short delay between patching the node capacity with a new resource and the first pod that requests the resource to be scheduled on that node.

Example:

Here is an example showing how to use `curl` to form an HTTP request that advertises five "foo" resources on node `k8s-node-1` whose master is `k8s-master`.

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-i
http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

Note: In the preceding request, `~1` is the encoding for the character `/` in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

To consume an opaque resource in a Pod, include the name of the opaque resource as a key in the `spec.containers[].resources.requests` map.

The Pod is scheduled only if all of the resource requests are satisfied, including cpu, memory and any opaque resources. The Pod will remain in the `PENDING` state as long as the resource request cannot be met by any node.

Example:

The Pod below requests 2 cpus and 1 "foo" (an opaque resource.)

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        pod.alpha.kubernetes.io/opaque-int-resource-foo: 1
```

Extended Resources

Kubernetes version 1.8 introduces Extended Resources. Extended Resources are fully-qualified resource names outside the `kubernetes.io` domain. Extended Resources allow cluster operators to advertise new node-level resources that would be otherwise unknown to the system. Extended Resource quantities must be integers and cannot be overcommitted.

Users can consume Extended Resources in Pod specs just like CPU and memory. The scheduler takes care of the resource accounting so that no more than the available amount is simultaneously allocated to Pods.

The API server restricts quantities of Extended Resources to whole numbers. Examples of *valid* quantities are `3`, `3000m` and `3Ki`. Examples of *invalid* quantities are `0.5` and `1500m`.

> **Note:** Extended Resources replace Opaque Integer Resources. Users can use any domain name prefix outside of the `kubernetes.io/` domain instead of the previous `pod.alpha.kubernetes.io/opaque-int-resource-` prefix.

There are two steps required to use Extended Resources. First, the cluster operator must advertise a per-node Extended Resource on one or more nodes. Second, users must request the Extended Resource in Pods.

To advertise a new Extended Resource, the cluster operator should submit a `PATCH` HTTP request to the API server to specify the available quantity in the `status.capacity` for a node in the cluster. After this operation, the node's `status.capacity` will include a new resource. The `status.allocatable` field is updated automatically with the new resource asynchronously by the kubelet. Note that because the scheduler uses the node `status.allocatable` value when evaluating Pod fitness, there may be a short delay between patching the node capacity with a new resource and the first pod that requests the resource to be scheduled on that node.

Example:

Here is an example showing how to use `curl` to form an HTTP request that advertises five "example.com/foo" resources on node `k8s-node-1` whose master is `k8s-master`.

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"}]
http://k8s-master:8080/api/v1/nodes/k8s-node-1/status
```

Note: In the preceding request, `~1` is the encoding for the character `/` in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

To consume an Extended Resource in a Pod, include the resource name as a key in the `spec.containers[].resources.requests` map.

Note: Extended resources cannot be overcommitted, so request and limit must be equal if both are present in a container spec.

The Pod is scheduled only if all of the resource requests are satisfied, including cpu, memory and any Extended Resources. The Pod will remain in the `PENDING` state as long as the resource request cannot be met by any node.

Example:

The Pod below requests 2 cpus and 1 "example.com/foo" (an extended resource.)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        example.com/foo: 1
```

Planned Improvements

Kubernetes version 1.5 only allows resource quantities to be specified on a Container. It is planned to improve accounting for resources that are shared by all Containers in a Pod, such as emptyDir volumes.

Kubernetes version 1.5 only supports Container requests and limits for CPU and memory. It is planned to add new resource types, including a node disk space resource, and a framework for adding custom resource types.

Kubernetes supports overcommitment of resources by supporting multiple levels of Quality of Service.

In Kubernetes version 1.5, one unit of CPU means different things on different cloud providers, and on different machine types within the same cloud providers. For example, on AWS, the capacity of a node is reported in ECUs, while in GCE it is reported in logical cores. We plan to revise the definition of the cpu resource to allow for more consistency across providers and platforms.

What's next

- Get hands-on experience assigning CPU and RAM resources to a container.
- Container
- ResourceRequirements

Assigning Pods to Nodes

You can constrain a pod to only be able to run on particular nodes or to prefer to run on particular nodes. There are several ways to do this, and they all use label selectors to make the selection. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want more control on a node where a pod lands, e.g. to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

You can find all the files for these examples in our docs repo here.

nodeSelector

`nodeSelector` is the simplest form of constraint. `nodeSelector` is a field of PodSpec. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

Let’s walk through an example of how to use `nodeSelector`.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of Kubernetes pods and that you have turned up a Kubernetes cluster.

Step One: Attach label to the node

Run `kubectl get nodes` to get the names of your cluster's nodes. Pick out the one that you want to add a label to, and then run `kubectl label nodes <node-name> <label-key>=<label-value>` to add a label to the node you've chosen. For example, if my node name is 'kubernetes-foo-node-1.c.a-robinson.internal' and my desired label is 'disktype=ssd', then I can run `kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd`.

If this fails with an "invalid command" error, you're likely using an older version of kubectl that doesn't have the `label` command. In that case, see the previous version of this guide for instructions on how to manually set labels on a node.

Also, note that label keys must be in the form of DNS labels (as described in the identifiers doc), meaning that they are not allowed to contain any upper-case letters.

You can verify that it worked by re-running `kubectl get nodes --show-labels` and checking that the node now has a label.

Step Two: Add a nodeSelector field to your pod configuration

Take whatever pod config file you want to run, and add a nodeSelector section to it, like this. For example, if this is my pod config:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
```

Then add a nodeSelector like so:

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

When you then run `kubectl create -f pod.yaml`, the pod will get scheduled on the node that you attached the label to! You can verify that it worked by running `kubectl get pods -o wide` and looking at the “NODE” that the pod was assigned to.

Interlude: built-in node labels

In addition to labels you attach, nodes come pre-populated with a standard set of labels. As of Kubernetes v1.4 these labels are

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `beta.kubernetes.io/os`
- `beta.kubernetes.io/arch`

Affinity and anti-affinity

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, currently in beta, greatly expands the types of constraints you can express. The key enhancements are

1. the language is more expressive (not just "AND of exact match")
2. you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled
3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

The affinity feature consists of two types of affinity, "node affinity" and "inter-pod affinity/anti-affinity." Node affinity is like the existing `nodeSelector` (but with the first two benefits listed above), while inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

`nodeSelector` continues to work as usual, but will eventually be deprecated, as node affinity can express everything that `nodeSelector` can express.

Node affinity (beta feature)

Node affinity was introduced as alpha in Kubernetes 1.2. Node affinity is conceptually similar to `nodeSelector` – it allows you to constrain which nodes your pod is eligible to schedule on, based on labels on the node.

There are currently two types of node affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that *must* be met for a pod to schedule onto a node (just like `nodeSelector` but using a more expressive syntax), while the latter specifies *preferences* that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how `nodeSelector` works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node. In the future we plan to offer `requiredDuringSchedulingRequiredDuringExecution` which will be just like `requiredDuringSchedulingIgnoredDuringExecution` except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements.

Thus an example of `requiredDuringSchedulingIgnoredDuringExecution` would be "only run the pod on nodes with Intel CPUs" and an example `preferredDuringSchedulingIgnoredDuringExecution` would be "try to run this set of pods in availability zone XYZ, but if it's not possible, then allow some to run elsewhere".

Node affinity is specified as field `nodeAffinity` of field `affinity` in the PodSpec.

Here's an example of a pod that uses node affinity:

pod-with-node-affinity.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: gcr.io/google_containers/pause:2.0
```

This node affinity rule says the pod can only be placed on a node with a label whose key is `kubernetes.io/e2e-az-name` and whose value is either `e2e-az1` or `e2e-az2`. In addition, among nodes that meet that criteria, nodes with a label whose key is `another-node-label-key` and whose value is `another-node-label-value` should be preferred.

You can see the operator `In` being used in the example. The new node affinity syntax supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`. There is no explicit "node anti-affinity" concept, but `NotIn` and `DoesNotExist` give that behavior.

If you specify both `nodeSelector` and `nodeAffinity`, *both* must be satisfied for the pod to be scheduled onto a candidate node.

If you specify multiple `nodeSelectorTerms` associated with `nodeAffinity` types, then the pod can be scheduled onto a node **if one of** the `nodeSelectorTerms` is satisfied.

If you specify multiple `matchExpressions` associated with `nodeSelectorTerms`, then the pod can be scheduled onto a node **only if all** `matchExpressions` can be satisfied.

If you remove or change the label of the node where the pod is scheduled, the pod won't be removed. In other words, the affinity selection works only at the time of scheduling the pod.

For more information on node affinity, see the design doc here.

Inter-pod affinity and anti-affinity (beta feature)

Inter-pod affinity and anti-affinity were introduced in Kubernetes 1.4. Inter-pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled *based on labels on pods that are already running on the node* rather than based on labels on nodes. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y." Y is expressed as a LabelSelector with an associated list of namespaces (or "all" namespaces); unlike nodes, because pods are namespaced (and therefore the labels on pods are implicitly namespaced), a label selector over pod labels must specify which namespaces the selector should apply to. Conceptually X is a topology domain like node, rack, cloud provider zone, cloud provider region, etc. You express it using a `topologyKey` which is the key for the node label that the system uses to denote such a topology domain, e.g. see the label keys listed above in the section Interlude: built-in node labels.

As with node affinity, there are currently two types of pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution` which denote "hard" vs. "soft" requirements. See the description in the node affinity section earlier. An example of `requiredDuringSchedulingIgnoredDuringExecution` affinity would be "co-locate the pods of service A and service B in the same zone, since they communicate a lot with each other" and an example `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity would be "spread the pods from this service across zones" (a hard requirement wouldn't make sense, since you probably have more pods than zones).

Inter-pod affinity is specified as field `podAffinity` of field `affinity` in the PodSpec. And inter-pod anti-affinity is specified as field `podAntiAffinity` of field `affinity` in the PodSpec.

An example of a pod that uses pod affinity:

pod-with-pod-affinity.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: gcr.io/google_containers/pause:2.0
```

The affinity on this pod defines one pod affinity rule and one pod anti-affinity rule. In this example, the `podAffinity` is `requiredDuringSchedulingIgnoredDuringExecution` while the `podAntiAffinity` is `preferredDuringSchedulingIgnoredDuringExecution`. The pod affinity rule says that the pod can schedule onto a node only if that node is in the same zone as at least one already-running pod that has a label with key "security" and value "S1". (More precisely, the pod is eligible to run on node N if node N has a label with key `failure-domain.beta.kubernetes.io/zone` and some value V such that there is at least one node in the cluster with key `failure-domain.beta.kubernetes.io/zone` and value V that is running a pod that has a label with key "security" and value "S1".) The pod anti-affinity rule says that the pod prefers to not schedule onto a node if that node is already running a pod with label having key "security" and value "S2". (If the `topologyKey` were `failure-domain.beta.kubernetes.io/zone` then it would mean that the pod cannot schedule onto a node if that node is in the same zone as a pod with label having key "security" and value "S2".) See the design doc. for many more examples of pod affinity and anti-affinity, both the `requiredDuringSchedulingIgnoredDuringExecution` flavor and the `preferredDuringSchedulingIgnoredDuringExecution` flavor.

The legal operators for pod affinity and anti-affinity are `In`, `NotIn`, `Exists`, `DoesNotExist`.

In principle, the `topologyKey` can be any legal label-key. However, for performance and security reasons, there are some constraints on topologyKey:

1. For affinity and for `RequiredDuringScheduling` pod anti-affinity, empty `topologyKey` is not allowed.
2. For `RequiredDuringScheduling` pod anti-affinity, the admission controller `LimitPodHardAntiAffinityTopology` was introduced to limit `topologyKey` to `kubernetes.io/hostname`. If you want to make it available for custom topologies, you may modify the admission controller, or simply disable it.
3. For `PreferredDuringScheduling` pod anti-affinity, empty `topologyKey` is interpreted as "all topologies" ("all topologies" here is now limited to the combination of `kubernetes.io/hostname`, `failure-domain.beta.kubernetes.io/zone` and `failure-domain.beta.kubernetes.io/region`).

4. Except for the above cases, the `topologyKey` can be any legal label-key.

In addition to `labelSelector` and `topologyKey`, you can optionally specify a list `namespaces` of namespaces which the `labelSelector` should match against (this goes at the same level of the definition as `labelSelector` and `topologyKey`). If omitted, it defaults to the namespace of the pod where the affinity/anti-affinity definition appears. If defined but empty, it means "all namespaces."

All `matchExpressions` associated with `requiredDuringSchedulingIgnoredDuringExecution` affinity and anti-affinity must be satisfied for the pod to schedule onto a node.

More Practical Use-cases

Interpod Affinity and AnitAffinity can be even more useful when they are used with higher level collections such as ReplicaSets, Statefulsets, Deployments, etc. One can easily configure that a set of workloads should be co-located in the same defined topology, eg., the same node.

Always co-located in the same node

In a three node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible. Here is the yaml snippet of a simple redis deployment with three replicas and selector label `app=store`

```
apiVersion: apps/v1beta1 # for versions before 1.6.0 use extensions/v1beta1
kind: Deployment
metadata:
  name: redis-cache
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      containers:
      - name: redis-server
        image: redis:3.2-alpine
```

Below yaml snippet of the webserver deployment has `podAffinity` configured, this informs the scheduler that all its replicas are to be co-located with pods that has selector label `app=store`

```
apiVersion: apps/v1beta1 # for versions before 1.6.0 use extensions/v1beta1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-app
```

if we create the above two deployments, our three node cluster could look like below.

node-1	node-2	node-3
webserver-1	*webserver-2*	*webserver-3*
cache-1	*cache-2*	*cache-3*

As you can see, all the 3 replicas of the `web-server` are automatically co-located with the cache as expected.

```
$kubectl get pods -o wide
NAME                             READY     STATUS    RESTARTS   AGE       IP
redis-cache-1450370735-6dzlj     1/1       Running   0          8m        10.192.4.2
redis-cache-1450370735-j2j96     1/1       Running   0          8m        10.192.2.2
redis-cache-1450370735-z73mh     1/1       Running   0          8m        10.192.3.1
web-server-1287567482-5d4dz      1/1       Running   0          7m        10.192.2.3
web-server-1287567482-6f7v5      1/1       Running   0          7m        10.192.4.3
web-server-1287567482-s330j      1/1       Running   0          7m        10.192.3.2
```

Best practice is to configure these highly available stateful workloads such as redis with AntiAffinity rules for more guaranteed spreading, which we will see in the next section.

Never co-located in the same node

Highly Available database statefulset has one master and three replicas, one may prefer none of the database instances to be co-located in the same node.

node-1	node-2	node-3	node-4
DB-MASTER	*DB-REPLICA-1*	*DB-REPLICA-2*	*DB-REPLICA-3*

Here is an example of zookeper statefulset configured with anti-affinity for high availability.

For more information on inter-pod affinity/anti-affinity, see the design doc here.

You may want to check Taints as well, which allow a *node* to *repel* a set of pods.

Taints and Tolerations

Node affinity, described here, is a property of *pods* that *attracts* them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite – they allow a *node* to *repel* a set of pods.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints. Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key=value:NoSchedule
```

places a taint on node `node1`. The taint has key `key`, value `value`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration. You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key"
  operator: "Exists"
  effect: "NoSchedule"
```

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

- the `operator` is `Exists` (in which case no `value` should be specified), or
- the `operator` is `Equal` and the `value` s are equal

`Operator` defaults to `Equal` if not specified.

NOTE: There are two special cases:

- An empty `key` with operator `Exists` matches all keys, values and effects which means this will tolerate everything.

```
tolerations:
- operator: "Exists"
```

- An empty `effect` matches all effects with key `key`.

```
tolerations:
- key: "key"
  operator: "Exists"
```

The above example used `effect` of `NoSchedule`. Alternatively, you can use `effect` of `PreferNoSchedule`. This is a "preference" or "soft" version of `NoSchedule` – the system will *try* to avoid placing a pod that does not tolerate the taint on the node, but it is not required. The third kind of `effect` is `NoExecute`, described later.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect `NoSchedule` then Kubernetes will not schedule the pod onto that node
- if there is no un-ignored taint with effect `NoSchedule` but there is at least one un-ignored taint with effect `PreferNoSchedule` then Kubernetes will *try* to not schedule the pod onto the node

- if there is at least one un-ignored taint with effect `NoExecute` then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect `NoExecute` is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and any pods that do tolerate the taint will never be evicted. However, a toleration with `NoExecute` effect can specify an optional `tolerationSeconds` field that dictates how long the pod will stay bound to the node after the taint is added. For example,

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

Example Use Cases

Taints and tolerations are a flexible way to steer pods *away* from nodes or evict pods that shouldn't be running. A few of the use cases are

- **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom admission controller). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them *and* ensure they *only* use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.
- **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom admission controller). For example, the admission controller could use some characteristic(s) of the pod to determine that the pod should be allowed to use the special nodes and hence the admission controller should add the toleration. To ensure that the pods that need the special hardware *only* schedule onto the nodes that have the special hardware, you will need some additional mechanism, e.g. you could represent the special resource using opaque integer resources and request it as a

resource in the PodSpec, or you could label the nodes that have the special hardware and use node affinity on the pods that need the hardware.

- **Taint based Evictions (alpha feature)**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

Earlier we mentioned the `NoExecute` taint effect, which affects pods that are already running on the node as follows

- pods that do not tolerate the taint are evicted immediately
- pods that tolerate the taint without specifying `tolerationSeconds` in their toleration specification remain bound forever
- pods that tolerate the taint with a specified `tolerationSeconds` remain bound for the specified amount of time

The above behavior is a beta feature. In addition, Kubernetes 1.6 has alpha support for representing node problems. In other words, the node controller automatically taints a node when certain condition is true. The built-in taints currently include:

- `node.alpha.kubernetes.io/notReady` : Node is not ready. This corresponds to the NodeCondition `Ready` being " `False` ".
- `node.alpha.kubernetes.io/unreachable` : Node is unreachable from the node controller. This corresponds to the NodeCondition `Ready` being " `Unknown` ".
- `node.kubernetes.io/outOfDisk` : Node becomes out of disk.
- `node.kubernetes.io/memoryPressure` : Node has memory pressure.
- `node.kubernetes.io/diskPressure` : Node has disk pressure.
- `node.kubernetes.io/networkUnavailable` : Node's network is unavailable.

- `node.cloudprovider.kubernetes.io/uninitialized`: When kubelet is started with "external" cloud provider, it sets this taint on a node to mark it as unusable. When a controller from the cloud-controller-manager initializes this node, kubelet removes this taint.

When the `TaintBasedEvictions` alpha feature is enabled (you can do this by including `TaintBasedEvictions=true` in `--feature-gates`, such as `--feature-gates=FooBar=true,TaintBasedEvictions=true`), the taints are automatically added by the NodeController (or kubelet) and the normal logic for evicting pods from nodes based on the Ready NodeCondition is disabled. (Note: To maintain the existing rate limiting behavior of pod evictions due to node problems, the system actually adds the taints in a rate-limited way. This prevents massive pod evictions in scenarios such as the master becoming partitioned from the nodes.) This alpha feature, in combination with `tolerationSeconds`, allows a pod to specify how long it should stay bound to a node that has one or both of these problems.

For example, an application with a lot of local state might want to stay bound to node for a long time in the event of network partition, in the hope that the partition will recover and thus the pod eviction can be avoided. The toleration the pod would use in that case would look like

```
tolerations:
- key: "node.alpha.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note that Kubernetes automatically adds a toleration for `node.alpha.kubernetes.io/notReady` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.alpha.kubernetes.io/notReady`. Likewise it adds a toleration for `node.alpha.kubernetes.io/unreachable` with `tolerationSeconds=300` unless the pod configuration provided by the user already has a toleration for `node.alpha.kubernetes.io/unreachable`.

These automatically-added tolerations ensure that the default pod behavior of remaining bound for 5 minutes after one of these problems is detected is maintained. The two default tolerations are added by the DefaultTolerationSeconds admission controller.

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.alpha.kubernetes.io/unreachable`
- `node.alpha.kubernetes.io/notReady`

This ensures that DaemonSet pods are never evicted due to these problems, which matches the behavior when this feature is disabled.

Taint Nodes by Condition

Version 1.8 introduces an alpha feature that causes the node controller to create taints corresponding to Node conditions. When this feature is enabled, the scheduler does not check conditions; instead the scheduler checks taints. This assures that conditions don't affect what's scheduled onto the Node. The user can choose to ignore some of the Node's problems (represented as conditions) by adding appropriate Pod tolerations.

To make sure that turning on this feature doesn't break DaemonSets, starting in version 1.8, the DaemonSet controller automatically adds the following `NoSchedule` tolerations to all daemons:

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk` (*only for critical pods*)

The above settings ensure backward compatibility, but we understand they may not fit all user's needs, which is why cluster admin may choose to add arbitrary tolerations to DaemonSets.

Secrets

Objects of type `secret` are intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a `secret` is safer and more flexible than putting it verbatim in a `pod` definition or in a docker image. See Secrets design document for more information.

Overview of Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: as files in a volume mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

Built-in Secrets

Service Accounts Automatically Create and Attach Secrets with API Credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and it automatically modifies your pods to use this type of secret.

The automatic creation and use of API credentials can be disabled or overridden if desired. However, if all you need to do is securely access the apiserver, this is the recommended workflow.

See the Service Account documentation for more information on how Service Accounts work.

Creating your own Secrets

Creating a Secret Using kubectl create secret

Say that some pods need to access a database. The username and password that the pods should use is in the files `./username.txt` and `./password.txt` on your local machine.

```
# Create files needed for rest of example.
$ echo -n "admin" > ./username.txt
$ echo -n "1f2d1e2e67df" > ./password.txt
```

The `kubectl create secret` command packages these files into a Secret and creates the object on the Apiserver.

```
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-fil
secret "db-user-pass" created
```

You can check that the secret was created like this:

```
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
db-user-pass          Opaque                                2         51s

$ kubectl describe secrets/db-user-pass
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

Note that neither `get` nor `describe` shows the contents of the file by default. This is to protect the secret from being exposed accidentally to someone looking or from being stored in a terminal log.

See decoding a secret for how to see the contents.

Creating a Secret Manually

You can also create a secret object in a file first, in json or yaml format, and then create that object.

Each item must be base64 encoded:

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

Now write a secret object that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

The data field is a map. Its keys must match `DNS_SUBDOMAIN`, except that leading dots are also allowed. The values are arbitrary data, encoded using base64.

Create the secret using `kubectl create`:

```
$ kubectl create -f ./secret.yaml
secret "mysecret" created
```

Encoding Note: The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/OS X users should avoid using the `-b` option to split long lines. Conversely Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if `-w` option is not available.

Decoding a Secret

Secrets can be retrieved via the `kubectl get secret` command. For example, to retrieve the secret created in the previous section:

```
$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

Decode the password field:

```
$ echo "MWYyZDFlMmU2N2Rm" | base64 --decode
1f2d1e2e67df
```

Using Secrets

Secrets can be mounted as data volumes or be exposed as environment variables to be used by a container in a pod. They can also be used by other parts of the system, without being directly exposed to the pod. For example, they can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Using Secrets as Files from a Pod

To consume a Secret in a volume in a Pod:

1. Create a secret or use an existing one. Multiple pods can reference the same secret.
2. Modify your Pod definition to add a volume under `spec.volumes[]`. Name the volume anything, and have a `spec.volumes[].secret.secretName` field equal to the name of the secret object.
3. Add a `spec.containers[].volumeMounts[]` to each container that needs the secret. Specify `spec.containers[].volumeMounts[].readOnly = true` and `spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the secrets to appear.

4. Modify your image and/or command line so that the program looks for files in that directory. Each key in the secret `data` map becomes the filename under `mountPath`.

This is an example of a pod that mounts a secret in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

Each secret you want to use needs to be referred to in `spec.volumes`.

If there are multiple containers in the pod, then each container needs its own `volumeMounts` block, but only one `spec.volumes` is needed per secret.

You can package many files into one secret, or use many secrets, whichever is convenient.

Projection of secret keys to specific paths

We can also control the paths within the volume where Secret keys are projected. You can use `spec.volumes[].secret.items` field to change target path of each key:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

What will happen:

- `username` secret is stored under `/etc/foo/my-group/my-username` file instead of `/etc/foo/username`.
- `password` secret is not projected

If `spec.volumes[].secret.items` is used, only keys specified in `items` are projected. To consume all keys from the secret, all of them must be listed in the `items` field. All listed keys must exist in the corresponding secret. Otherwise, the volume is not created.

Secret files permissions

You can also specify the permission mode bits files part of a secret will have. If you don't specify any, `0644` is used by default. You can specify a default mode for the whole secret volume and override per key if needed.

For example, you can specify a default mode like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      defaultMode: 256
```

Then, the secret will be mounted on `/etc/foo` and all the files created by the secret volume mount will have permission `0400`.

Note that the JSON spec doesn't support octal notation, so use the value 256 for 0400 permissions. If you use yaml instead of json for the pod, you can use octal notation to specify permissions in a more natural way.

You can also use mapping, as in the previous example, and specify different permission for different files like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. Owing to JSON limitations, you must specify the mode in decimal notation.

Note that this permission value might be displayed in decimal notation if you read it later.

Consuming Secret Values from Volumes

Inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
$ ls /etc/foo/
username
password
$ cat /etc/foo/username
admin
$ cat /etc/foo/password
1f2d1e2e67df
```

The program in a container is responsible for reading the secrets from the files.

Mounted Secrets are updated automatically

When a secret being already consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted secret is fresh on every periodic sync. However, it is using its local ttl-based cache for getting the current value of the secret. As a result, the total delay from the moment when the secret is updated to the moment when new keys are projected to the pod can be as long as kubelet sync period + ttl of secrets cache in kubelet.

Using Secrets as Environment Variables

To use a secret in an environment variable in a pod:

1. Create a secret or use an existing one. Multiple pods can reference the same secret.
2. Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in `env[x].valueFrom.secretKeyRef`.

3. Modify your image and/or command line so that the program looks for values in the specified environment variables

This is an example of a pod that uses secrets from environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Consuming Secret Values from Environment Variables

Inside a container that consumes a secret in an environment variables, the secret keys appear as normal environment variables containing the base-64 decoded values of the secret data. This is the result of commands executed inside the container from the example above:

```
$ echo $SECRET_USERNAME
admin
$ echo $SECRET_PASSWORD
1f2d1e2e67df
```

Using imagePullSecrets

An imagePullSecret is a way to pass a secret that contains a Docker (or other) image registry password to the Kubelet so it can pull a private image on behalf of your Pod.

Manually specifying an imagePullSecret

Use of imagePullSecrets is described in the images documentation

Arranging for imagePullSecrets to be Automatically Attached

You can manually create an imagePullSecret, and reference it from a serviceAccount. Any pods created with that serviceAccount or that default to use that serviceAccount, will get their imagePullSecret field set to that of the service account. See Adding ImagePullSecrets to a service account for a detailed explanation of that process.

Automatic Mounting of Manually Created Secrets

Manually created secrets (e.g. one containing a token for accessing a github account) can be automatically attached to pods based on their service account. See Injecting Information into Pods Using a PodPreset for a detailed explanation of that process.

Details

Restrictions

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type `Secret`. Therefore, a secret needs to be created before any pods that depend on it.

Secret API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage creation of very large secrets which would exhaust apiserver and kubelet memory. However, creation of many smaller secrets could also exhaust memory. More comprehensive limits on memory usage due to secrets is a planned feature.

Kubelet only supports use of secrets for Pods it gets from the API server. This includes any pods created using kubectl, or indirectly via a replication controller. It does not include pods created via the kubelets `--manifest-url` flag, its `--config` flag, or its REST API (these are not common ways to create pods.)

Secrets must be created before they are consumed in pods as environment variables unless they are marked as optional. References to Secrets that do not exist will prevent the pod from starting.

References via `secretKeyRef` to keys that do not exist in a named Secret will prevent the pod from starting.

Secrets used to populate environment variables via `envFrom` that have keys that are considered invalid environment variable names will have those keys skipped. The pod will be allowed to start. There will be an event whose reason is `InvalidVariableNames` and the message will contain the list of invalid keys that were skipped. The example shows a pod which refers to the default/mysecret that contains 2 invalid keys, 1badkey and 2alsobad.

```
$ kubectl get events
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND      SUBOBJECT
0s         0s          1         dapi-test-pod   Pod
```

Secret and Pod Lifetime interaction

When a pod is created via the API, there is no check whether a referenced secret exists. Once a pod is scheduled, the kubelet will try to fetch the secret value. If the secret cannot be fetched because it does not exist or because of a temporary lack of connection to the API server, kubelet will periodically retry. It will report an event about the pod explaining the reason it is not started yet. Once the secret is fetched, the kubelet will create and mount a volume containing it. None of the pod's containers will start until all the pod's volumes are mounted.

Use cases

Use-Case: Pod with ssh keys

Create a secret containing some ssh keys:

```
$ kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/path/to
```

Security Note: think carefully before sending your own ssh keys: other users of the cluster may have access to the secret. Use a service account which you want to be accessible to all the users with whom you share the Kubernetes cluster, and can revoke if they are compromised.

Now we can create a pod which references the secret with the ssh key and consumes it in a volume:

```
kind: Pod
apiVersion: v1
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: ssh-key-secret
  containers:
  - name: ssh-test-container
    image: mySshImage
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

When the container's command runs, the pieces of the key will be available in:

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

The container is then free to use the secret data to establish an ssh connection.

Use-Case: Pods with prod / test credentials

This example illustrates a pod which consumes a secret containing prod credentials and another pod which consumes a secret with test environment credentials.

Make the secrets:

```
$ kubectl create secret generic prod-db-secret --from-literal=username=produser --
secret "prod-db-secret" created
$ kubectl create secret generic test-db-secret --from-literal=username=testuser --
secret "test-db-secret" created
```

Now make the pods:

```
apiVersion: v1
kind: List
items:
- kind: Pod
  apiVersion: v1
  metadata:
    name: prod-db-client-pod
    labels:
      name: prod-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: prod-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
- kind: Pod
  apiVersion: v1
  metadata:
    name: test-db-client-pod
    labels:
      name: test-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: test-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
```

Both containers will have the following files present on their filesystems with the values for each container's environment:

```
/etc/secret-volume/username
/etc/secret-volume/password
```

Note how the specs for the two pods differ only in one field; this facilitates creating pods with different capabilities from a common pod config template.

You could further simplify the base pod specification by using two Service Accounts: one called, say, `prod-user` with the `prod-db-secret`, and one called, say, `test-user` with the `test-db-secret`. Then, the pod spec can be shortened to, for example:

```
kind: Pod
apiVersion: v1
metadata:
  name: prod-db-client-pod
  labels:
    name: prod-db-client
spec:
  serviceAccount: prod-db-client
  containers:
  - name: db-client-container
    image: myClientImage
```

Use-case: Dotfiles in secret volume

In order to make piece of data ‘hidden’ (i.e., in a file whose name begins with a dot character), simply make that key begin with a dot. For example, when the following secret is mounted into a volume:

```
kind: Secret
apiVersion: v1
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
kind: Pod
apiVersion: v1
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
  - name: dotfile-test-container
    image: gcr.io/google_containers/busybox
    command:
    - ls
    - "-l"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

The `secret-volume` will contain a single file, called `.secret-file`, and the `dotfile-test-container` will have this file present at the path `/etc/secret-volume/.secret-file`.

NOTE

Files beginning with dot characters are hidden from the output of `ls -l`; you must use `ls -la` to see them when listing directory contents.

Use-case: Secret visible to one container in a pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (e.g. over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Best practices

Clients that use the secrets API

When deploying applications that interact with the secrets API, access should be limited using authorization policies such as RBAC.

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

For these reasons `watch` and `list` requests for secrets within a namespace are extremely powerful capabilities and should be avoided, since listing secrets allows the clients to inspect the values if all secrets are in that namespace. The ability to `watch` and `list` all secrets in a cluster should be reserved for only the most privileged, system-level components.

Applications that need to access the secrets API should perform `get` requests on the secrets they need. This lets administrators restrict access to all secrets while white-listing access to individual instances that the app needs.

For improved performance over a looping `get`, clients can design resources that reference a secret then `watch` the resource, re-requesting the secret when the reference changes. Additionally, a "bulk watch" API to let clients `watch` individual resources has also been proposed, and will likely be available in future releases of Kubernetes.

Security Properties

Protections

Because `secret` objects can be created independently of the `pods` that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing pods. The system can also take additional precautions with `secret` objects, such as avoiding writing them to disk where possible.

A secret is only sent to a node if a pod on that node requires it. It is not written to disk. It is stored in a tmpfs. It is deleted once the pod that depends on it is deleted.

On most Kubernetes-project-maintained distributions, communication between user to the apiserver, and from apiserver to the kubelets, is protected by SSL/TLS. Secrets are protected when transmitted over these channels.

Secret data on nodes is stored in tmpfs volumes and thus does not come to rest on the node.

There may be secrets for several pods on the same node. However, only the secrets that a pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the secrets of another pod.

There may be several containers in a pod. However, each container in a pod has to request the secret volume in its `volumeMounts` for it to be visible within the container. This can be used to construct useful security partitions at the Pod level.

Risks

- In the API server secret data is stored as plaintext in etcd; therefore:
  - Administrators should limit access to etcd to admin users
  - Secret data in the API server is at rest on the disk that etcd uses; admins may want to wipe/shred disks used by etcd when no longer in use
- If you configure the secret through a manifest (JSON or YAML) file which has the secret data encoded as base64, sharing this file or checking it in to a source repository means the secret is compromised. Base64 encoding is not an encryption method and is considered the same as plain text.
- Applications still need to protect the value of secret after reading it from the volume, such as not accidentally logging it or transmitting it to an untrusted party.

- A user who can create a pod that uses a secret can also see the value of that secret. Even if apiserver policy does not allow that user to read the secret object, the user could run a pod which exposes the secret.
- If multiple replicas of etcd are run, then the secrets will be shared between them. By default, etcd does not secure peer-to-peer communication with SSL/TLS, though this can be configured.
- Currently, anyone with root on any node can read any secret from the apiserver, by impersonating the kubelet. It is a planned feature to only send secrets to nodes that actually require them, to restrict the impact of a root exploit on a single node.

Organizing Cluster Access Using kubeconfig Files

Use kubeconfig files to organize information about clusters, users, namespaces, and authentication mechanisms. The `kubectl` command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster.

> **Note:** A file that is used to configure access to clusters is called a *kubeconfig file*. This is a generic way of referring to configuration files. It does not mean that there is a file named `kubeconfig`.

By default, `kubectl` looks for a file named `config` in the `$HOME/.kube` directory. You can specify other kubeconfig files by setting the `KUBECONFIG` environment variable or by setting the `--kubeconfig` flag.

For step-by-step instructions on creating and specifying kubeconfig files, see Configure Access to Multiple Clusters.

Supporting multiple clusters, users, and authentication mechanisms

Suppose you have several clusters, and your users and components authenticate in a variety of ways. For example:

- A running kubelet might authenticate using certificates.
- A user might authenticate using tokens.
- Administrators might have sets of certificates that they provide to individual users.

With kubeconfig files, you can organize your clusters, users, and namespaces. You can also define contexts to quickly and easily switch between clusters and namespaces.

Context

A *context* element in a kubeconfig file is used to group access parameters under a convenient name. Each context has three parameters: cluster, namespace, and user. By default, the `kubectl` command-line tool uses parameters from the *current context* to communicate with the cluster.

To choose the current context: `kubectl config use-context`

The KUBECONFIG environment variable

The `KUBECONFIG` environment variable holds a list of kubeconfig files. For Linux and Mac, the list is colon-delimited. For Windows, the list is semicolon-delimited. The `KUBECONFIG` environment variable is not required. If the `KUBECONFIG` environment variable doesn't exist, `kubectl` uses the default kubeconfig file, `$HOME/.kube/config`.

If the `KUBECONFIG` environment variable does exist, `kubectl` uses an effective configuration that is the result of merging the files listed in the `KUBECONFIG` evironment variable.

Merging kubeconfig files

To see your configuration, enter this command:

```
kubectl config view
```

As described previously, the output might be from a single kubeconfig file, or it might be the result of merging several kubeconfig files.

Here are the rules that `kubectl` uses when it merges kubeconfig files:

1. If the `--kubeconfig` flag is set, use only the specified file. Do not merge. Only one instance of this flag is allowed.

   Otherwise, if the `KUBECONFIG` environment variable is set, use it as a list of files that should be merged. Merge the files listed in the `KUBECONFIG` envrionment variable according to these rules:

   1. Ignore empty filenames.
   2. Produce errors for files with content that cannot be deserialized.
   3. The first file to set a particular value or map key wins.
   4. Never change the value or map key. Example: Preserve the context of the first file to set `current-context`. Example: If two files specify a `red-user`, use only values from the first file's `red-user`. Even if the second file has non-conflicting entries under `red-user`, discard them.

   For an example of setting the `KUBECONFIG` environment variable, see Setting the KUBECONFIG environment variable.

   Otherwise, use the default kubeconfig file, `$HOME/.kube/config`, with no merging.

2. Determine the context to use based on the first hit in this chain:

   1. Use the `--context` command-line flag if it exits.
   2. Use the `current-context` from the merged kubeconfig files.

   An empty context is allowed at this point.

3. Determine the cluster and user. At this point, there might or might not be a context. Determine the cluster and user based on the first hit in this chain, which is run twice: once for user and once for cluster:

   1. Use a command-line flag if it exists: `--user` or `--cluster`.
   2. If the context is non-empty, take the user or cluster from the context.

The user and cluster can be empty at this point.

4. Determine the actual cluster information to use. At this point, there might or might not be cluster information. Build each piece of the cluster information based on this chain; the first hit wins:
    1. Use command line flags if they exist: `--server`, `--certificate-authority`, `--insecure-skip-tls-verify`.
    2. If any cluster information attributes exist from the merged kubeconfig files, use them.
    3. If there is no server location, fail.
5. Determine the actual user information to use. Build user information using the same rules as cluster information, except allow only one authentication technique per user:
    1. Use command line flags if they exist: `--client-certificate`, `--client-key`, `--username`, `--password`, `--token`.
    2. Use the `user` fields from the merged kubeconfig files.
    3. If there are two conflicting techniques, fail.
6. For any information still missing, use default values and potentially prompt for authentication information.

File references

File and path references in a kubeconfig file are relative to the location of the kubeconfig file. File references on the command line are relative to the current working directory. In `$HOME/.kube/config`, relative paths are stored relatively, and absolute paths are stored absolutely.

What's next

- Configure Access to Multiple Clusters
- kubectl config

Pod Priority and Preemption

FEATURE STATE: `Kubernetes v1.8` alpha

Pods in Kubernetes 1.8 and later can have priority. Priority indicates the importance of a Pod relative to other Pods. When a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible. In a future Kubernetes release, priority will also affect out-of-resource eviction ordering on the Node.

> **Note:** Preemption does not respect PodDisruptionBudget; see the limitations section for more details.

How to use priority and preemption

To use priority and preemption in Kubernetes 1.8, follow these steps:

1. Enable the feature.

2. Add one or more PriorityClasses.

3. Create Pods with `PriorityClassName` set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add `PriorityClassName` to the Pod

template of a collection object like a Deployment.

The following sections provide more information about these steps.

Enabling priority and preemption

Pod priority and preemption is disabled by default in Kubernetes 1.8. To enable the feature, set this command-line flag for the API server and the scheduler:

```
--feature-gates=PodPriority=true
```

Also set this flag for API server:

```
--runtime-config=scheduling.k8s.io/v1alpha1=true
```

After the feature is enabled, you can create PriorityClasses and create Pods with `PriorityClassName` set.

If you try the feature and then decide to disable it, you must remove the PodPriority command-line flag or set it to false, and then restart the API server and scheduler. After the feature is disabled, the existing Pods keep their priority fields, but preemption is disabled, and priority fields are ignored, and you cannot set PriorityClassName in new Pods.

PriorityClass

A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the `name` field of the PriorityClass object's metadata. The value is specified in the required `value` field. The higher the value, the higher the priority.

A PriorityClass object can have any 32-bit integer value smaller than or equal to 1 billion. Larger numbers are reserved for critical system Pods that should not normally be preempted or evicted. A cluster admin should create one PriorityClass object for each such mapping that they want.

PriorityClass also has two optional fields: `globalDefault` and `description`. The `globalDefault` field indicates that the value of this PriorityClass should be used for Pods without a `PriorityClassName`. Only one PriorityClass with `globalDefault` set to true can exist in the system. If there is no PriorityClass with `globalDefault` set, the priority of Pods with no `PriorityClassName` is zero.

The `description` field is an arbitrary string. It is meant to tell users of the cluster when they should use this PriorityClass.

> **Note 1**: If you upgrade your existing cluster and enable this feature, the priority of your existing Pods will be considered to be zero.

> **Note 2**: Addition of a PriorityClass with `globalDefault` set to true does not change the priorities of existing Pods. The value of such a PriorityClass is used only for Pods created after the PriorityClass is added.

> **Note 3**: If you delete a PriorityClass, existing Pods that use the name of the deleted priority class remain unchanged, but you are not able to create more Pods that use the name of the deleted PriorityClass.

Example PriorityClass

```yaml
apiVersion: v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the `priorityClassName` field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let's call the pending pod P. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than P would enable P to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get deleted from the Node. After the Pods are gone, P can be scheduled on the Node.

Limitations of preemption (alpha version)

Starvation of preempting Pod

When Pods are preempted, the victims get their graceful termination period. They have that much time to finish their work and exit. If they don't, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod

(P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue, and one or more of them may be considered and scheduled to N before the scheduler considers scheduling P on N. In such a case, it is likely that when all the victims exit, Pod P won't fit on Node N anymore. So, scheduler will have to preempt other Pods on Node N or another Node so that P can be scheduled. This scenario might be repeated again for the second and subsequent rounds of preemption, and P might not get scheduled for a while. This scenario can cause problems in various clusters, but is particularly problematic in clusters with a high Pod creation rate.

We will address this problem in the beta version of Pod preemption. The solution we plan to implement is provided here.

PodDisruptionBudget is not supported

A Pod Disruption Budget (PDB) allows application owners to limit the number Pods of a replicated application that are down simultaneously from voluntary disruptions. However, the alpha version of preemption does not respect PDB when choosing preemption victims. We plan to add PDB support in beta, but even in beta, respecting PDB will be best effort. The Scheduler will try to find victims whose PDB won't be violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

Inter-Pod affinity on lower-priority Pods

In version 1.8, a Node is considered for preemption only when the answer to this question is yes: "If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending pod be scheduled on the Node?"

> **Note:** Preemption does not necessarily remove all lower-priority Pods. If the pending pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

We might address this issue in future versions, but we don't have a clear plan yet. We will not consider it a blocker for Beta or GA. Part of the reason is that finding the set of lower-priority Pods that satisfy all inter-Pod affinity rules is computationally expensive, and adds substantial complexity to the preemption logic. Besides, even if preemption keeps the lower-priority Pods to satisfy inter-Pod affinity, the lower priority Pods might be preempted later by other Pods, which removes the benefits of having the complex logic of respecting inter-Pod affinity.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority pods.

Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here's an example:

- Pod P is being considered for Node N.
- Pod Q is running on another Node in the same zone as Node N.
- Pod P has anit-affinity with Pod Q.
- There are no other cases of anti-affinity between Pod P and other Pods in the zone.
- In order to schedule Pod P on Node N, Pod Q should be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if we find an algorithm with reasonable performance. We cannot promise anything at this point, and cross Node preemption will not be considered a blocker for Beta or GA.

Services

Kubernetes `Pods` are mortal. They are born and when they die, they are not resurrected. `ReplicationControllers` in particular create and destroy `Pods` dynamically (e.g. when scaling up or down or when doing rolling updates). While each `Pod` gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of `Pods` (let's call them backends) provides functionality to other `Pods` (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set?

Enter `Services`.

A Kubernetes `Service` is an abstraction which defines a logical set of `Pods` and a policy by which to access them - sometimes called a micro-service. The set of `Pods` targeted by a `Service` is (usually) determined by a `Label Selector` (see below for why you might want a `Service` without a selector).

As an example, consider an image-processing backend which is running with 3 replicas. Those replicas are fungible - frontends do not care which backend they use. While the actual `Pods` that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The `Service` abstraction enables this decoupling.

For Kubernetes-native applications, Kubernetes offers a simple `Endpoints` API that is updated whenever the set of `Pods` in a `Service` changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend `Pods`.

Defining a service

A `Service` in Kubernetes is a REST object, similar to a `Pod`. Like all of the REST objects, a `Service` definition can be POSTed to the apiserver to create a new instance. For example, suppose you have a set of `Pods` that each expose port 9376 and carry a label `"app=MyApp"`.

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

This specification will create a new `Service` object named “my-service” which targets TCP port 9376 on any `Pod` with the `"app=MyApp"` label. This `Service` will also be assigned an IP address (sometimes called the “cluster IP”), which is used by the service proxies (see below). The `Service`’s selector will be evaluated continuously and the results will be POSTed to an `Endpoints` object also named “my-service”.

Note that a `Service` can map an incoming port to any `targetPort`. By default the `targetPort` will be set to the same value as the `port` field. Perhaps more interesting is that `targetPort` can be a string, referring to the name of a port in the backend `Pods`. The actual port number assigned to that name can be different in each backend `Pod`. This offers a lot of flexibility for deploying and evolving your `Services`. For example, you can change the port number that pods expose in the next version of your backend software, without breaking clients.

Kubernetes `Services` support `TCP` and `UDP` for protocols. The default is `TCP`.

Services without selectors

Services generally abstract access to Kubernetes `Pods`, but they can also abstract other kinds of backends. For example:

- You want to have an external database cluster in production, but in test you use your own databases.
- You want to point your service to a service in another `Namespace` or on another cluster.
- You are migrating your workload to Kubernetes and some of your backends run outside of Kubernetes.

In any of these scenarios you can define a service without a selector:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Because this service has no selector, the corresponding `Endpoints` object will not be created. You can manually map the service to your own specific endpoints:

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

NOTE: Endpoint IPs may not be loopback (127.0.0.0/8), link-local (169.254.0.0/16), or link-local multicast (224.0.0.0/24).

Accessing a `Service` without a selector works the same as if it had a selector. The traffic will be routed to endpoints defined by the user (`1.2.3.4:9376` in this example).

An ExternalName service is a special case of service that does not have selectors. It does not define any ports or endpoints. Rather, it serves as a way to return an alias to an external service residing outside the cluster.

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

When looking up the host `my-service.prod.svc.CLUSTER`, the cluster DNS service will return a `CNAME` record with the value `my.database.example.com`. Accessing such a service works in the same way as others, with the only difference that the redirection happens at the DNS level and no proxying or forwarding occurs. Should you later decide to move your database into your cluster, you can start its pods, add appropriate selectors or endpoints and change the service `type`.

Virtual IPs and service proxies

Every node in a Kubernetes cluster runs a `kube-proxy`. `kube-proxy` is responsible for implementing a form of virtual IP for `Services` of type other than `ExternalName`. In Kubernetes v1.0 the proxy was purely in userspace. In Kubernetes v1.1 an iptables proxy was added, but was not the default operating mode. Since Kubernetes v1.2, the iptables proxy is the default.

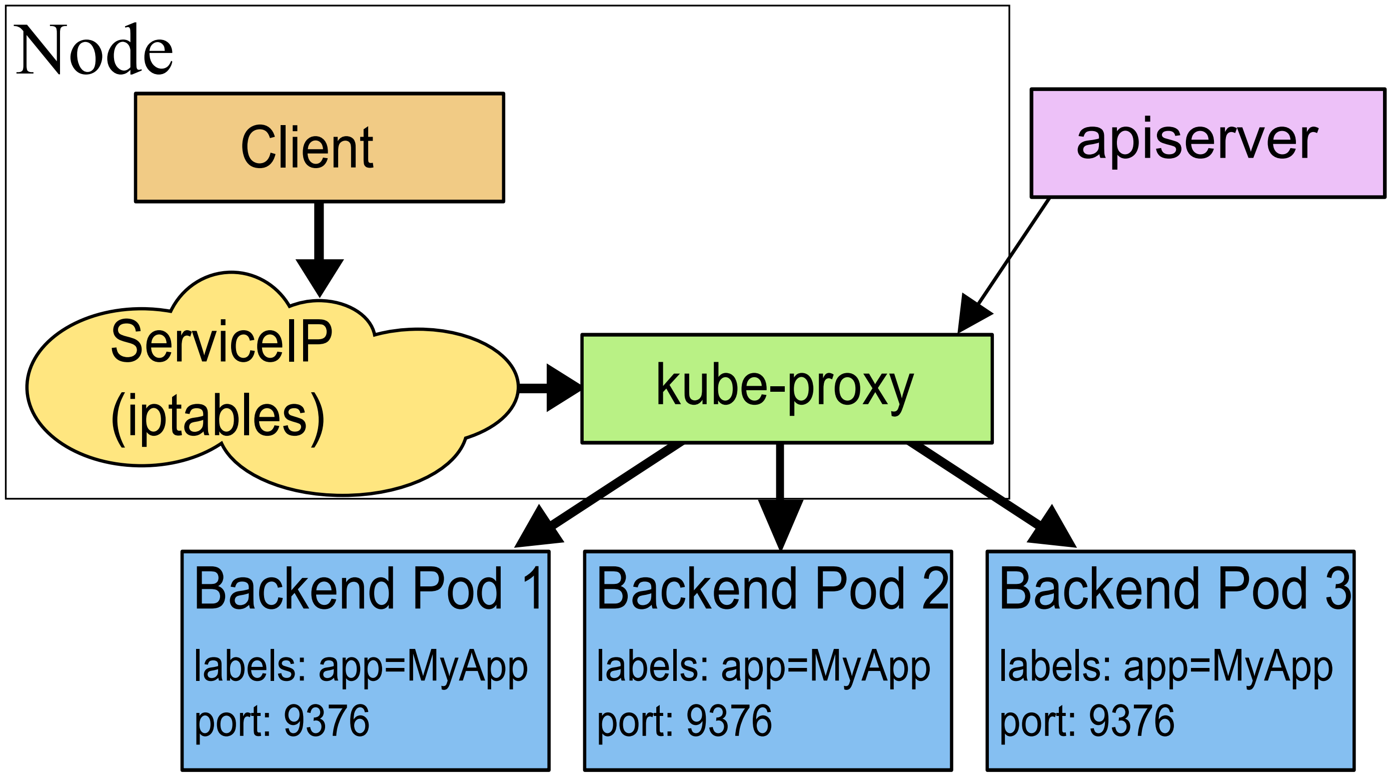
As of Kubernetes v1.0, `Services` are a “layer 4” (TCP/UDP over IP) construct. In Kubernetes v1.1 the `Ingress` API was added (beta) to represent “layer 7” (HTTP) services.

Proxy-mode: userspace

In this mode, kube-proxy watches the Kubernetes master for the addition and removal of `Service` and `Endpoints` objects. For each `Service` it opens a port (randomly chosen) on the local node. Any connections to this “proxy port” will be proxied to one of the `Service`’s backend `Pods` (as reported in `Endpoints`). Which backend `Pod` to use is decided based on the `SessionAffinity` of the `Service`. Lastly, it installs iptables rules which capture traffic to the `Service`’s `clusterIP` (which is virtual) and `Port` and redirects that traffic to the proxy port which proxies the backend `Pod`.

The net result is that any traffic bound for the `Service`’s IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or `Services` or `Pods`.

By default, the choice of backend is round robin. Client-IP based session affinity can be selected by setting `service.spec.sessionAffinity` to `"ClientIP"` (the default is `"None"`), and you can set the max session sticky time by setting the field `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` if you have already set `service.spec.sessionAffinity` to `"ClientIP"` (the default is "10800").

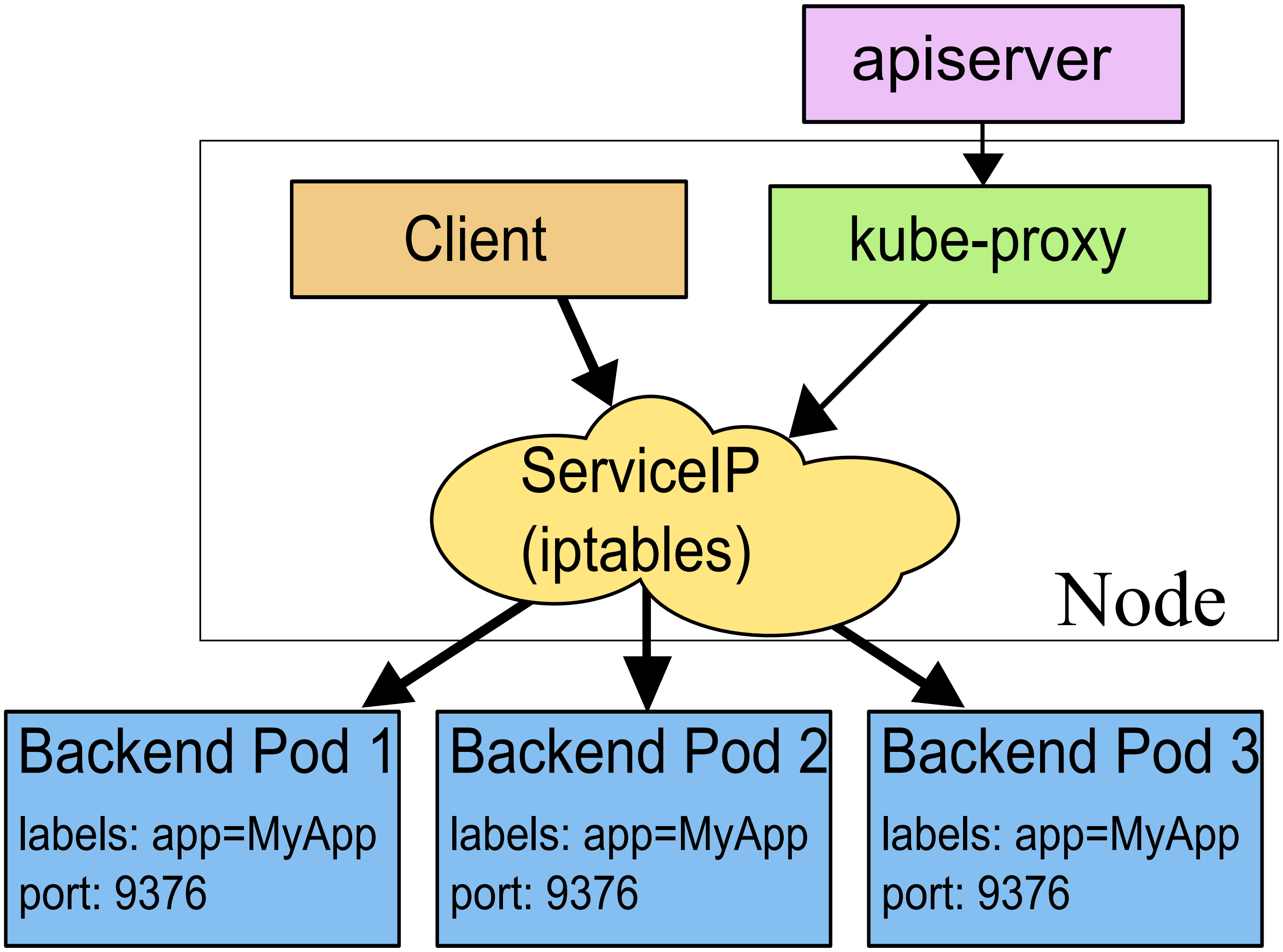


Proxy-mode: iptables

In this mode, kube-proxy watches the Kubernetes master for the addition and removal of `Service` and `Endpoints` objects. For each `Service` it installs iptables rules which capture traffic to the `Service`'s `clusterIP` (which is virtual) and `Port` and redirects that traffic to one of the `Service`'s backend sets. For each `Endpoints` object it installs iptables rules which select a backend `Pod`.

By default, the choice of backend is random. Client-IP based session affinity can be selected by setting `service.spec.sessionAffinity` to `"ClientIP"` (the default is `"None"`), and you can set the max session sticky time by setting the field `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` if you have already set `service.spec.sessionAffinity` to `"ClientIP"` (the default is "10800").

As with the userspace proxy, the net result is that any traffic bound for the `Service`'s IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or `Services` or `Pods`. This should be faster and more reliable than the userspace proxy. However, unlike the userspace proxier, the iptables proxier cannot automatically retry another `Pod` if the one it initially selects does not respond, so it depends on having working readiness probes.



Multi-Port Services

Many `Services` need to expose more than one port. For this case, Kubernetes supports multiple port definitions on a `Service` object. When using multiple ports you must give all of your ports names, so that endpoints can be disambiguated. For example:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
    port: 443
    targetPort: 9377
```

Choosing your own IP address

You can specify your own cluster IP address as part of a `Service` creation request. To do this, set the `spec.clusterIP` field. For example, if you already have an existing DNS entry that you wish to replace, or legacy systems that are configured for a specific IP address and difficult to re-configure. The IP address that a user chooses must be a valid IP address and within the `service-cluster-ip-range` CIDR range that is specified by flag to the API server. If the IP address value is invalid, the apiserver returns a 422 HTTP status code to indicate that the value is invalid.

Why not use round-robin DNS?

A question that pops up every now and then is why we do all this stuff with virtual IPs rather than just use standard round-robin DNS. There are a few reasons:

- There is a long history of DNS libraries not respecting DNS TTLs and caching the results of name lookups.
- Many apps do DNS lookups once and cache the results.
- Even if apps and libraries did proper re-resolution, the load of every client re-resolving DNS over and over would be difficult to manage.

We try to discourage users from doing things that hurt themselves. That said, if enough people ask for this, we may implement it as an alternative.

Discovering services

Kubernetes supports 2 primary modes of finding a `Service` - environment variables and DNS.

Environment variables

When a `Pod` is run on a `Node`, the kubelet adds a set of environment variables for each active `Service`. It supports both Docker links compatible variables (see makeLinkVariables) and simpler `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores.

For example, the Service `"redis-master"` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11 produces the following environment variables:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

This does imply an ordering requirement - any `Service` that a `Pod` wants to access must be created before the `Pod` itself, or else the environment variables will not be populated. DNS does not have this restriction.

DNS

An optional (though strongly recommended) cluster add-on is a DNS server. The DNS server watches the Kubernetes API for new `Services` and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all `Pods` should be able to do name resolution of `Services` automatically.

For example, if you have a `Service` called `"my-service"` in Kubernetes `Namespace` `"my-ns"` a DNS record for `"my-service.my-ns"` is created. `Pods` which exist in the `"my-ns"` `Namespace` should be able to find it by simply doing a name lookup for `"my-service"`. `Pods` which exist in other `Namespaces` must qualify the name as `"my-service.my-ns"`. The result of these name lookups is the cluster IP.

Kubernetes also supports DNS SRV (service) records for named ports. If the `"my-service.my-ns"` `Service` has a port named `"http"` with protocol `TCP`, you can do a DNS SRV query for `"_http._tcp.my-service.my-ns"` to discover the port number for `"http"`.

The Kubernetes DNS server is the only way to access services of type `ExternalName`. More information is available in the DNS Pods and Services.

Headless services

Sometimes you don’t need or want load-balancing and a single service IP. In this case, you can create “headless” services by specifying `"None"` for the cluster IP (`spec.clusterIP`).

This option allows developers to reduce coupling to the Kubernetes system by allowing them freedom to do discovery their own way. Applications can still use a self-registration pattern and adapters for other discovery systems could easily be built upon this API.

For such `Services`, a cluster IP is not allocated, kube-proxy does not handle these services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the service has selectors defined.

With selectors

For headless services that define selectors, the endpoints controller creates `Endpoints` records in the API, and modifies the DNS configuration to return A records (addresses) that point directly to the `Pods` backing the `Service`.

Without selectors

For headless services that do not define selectors, the endpoints controller does not create `Endpoints` records. However, the DNS system looks for and configures either:

- CNAME records for `ExternalName`-type services.
- A records for any `Endpoints` that share a name with the service, for all other types.

Publishing services - service types

For some parts of your application (e.g. frontends) you may want to expose a Service onto an external (outside of your cluster) IP address.

Kubernetes `ServiceTypes` allow you to specify what kind of service you want. The default is `ClusterIP`.

`Type` values and their behaviors are:

- `ClusterIP`: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default `ServiceType`.
- `NodePort`: Exposes the service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` service, to which the NodePort service will route, is automatically created. You'll be able to contact the `NodePort` service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.
- `LoadBalancer`: Exposes the service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` services, to which the external load balancer will route, are automatically created.
- `ExternalName`: Maps the service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of `kube-dns`.

Type NodePort

If you set the `type` field to `"NodePort"`, the Kubernetes master will allocate a port from a flag-configured range (default: 30000-32767), and each Node will proxy that port (the same port number on every Node) into your `Service`. That port will be reported in your `Service`'s `spec.ports[*].nodePort` field.

If you want a specific port number, you can specify a value in the `nodePort` field, and the system will allocate you that port or else the API transaction will fail (i.e. you need to take care about possible port collisions yourself). The value you specify must be in the configured range for node ports.

This gives developers the freedom to set up their own load balancers, to configure environments that are not fully supported by Kubernetes, or even to just expose one or more nodes' IPs directly.

Note that this Service will be visible as both `<NodeIP>:spec.ports[*].nodePort` and `spec.clusterIp:spec.ports[*].port`.

Type LoadBalancer

On cloud providers which support external load balancers, setting the `type` field to `"LoadBalancer"` will provision a load balancer for your `Service`. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer will be published in the `Service`'s `status.loadBalancer` field. For example:

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
    nodePort: 30061
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

Traffic from the external load balancer will be directed at the backend `Pods`, though exactly how that works depends on the cloud provider. Some cloud providers allow the `loadBalancerIP` to be specified. In those cases, the load-balancer will be created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, an ephemeral IP will be assigned to the loadBalancer. If the `loadBalancerIP` is specified, but the cloud provider does not support the feature, the field will be ignored.

Special notes for Azure: To use user-specified public type `loadBalancerIP`, a static type public IP address resource needs to be created first, and it should be in the same resource group of the cluster. Then you could specify the assigned IP address as `loadBalancerIP`.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from services inside the same VPC.

In a split-horizon DNS environment you would need two services to be able to route both external and internal traffic to your endpoints.

This can be achieved by adding the following annotations to the service based on cloud provider.

Default GCP AWS Azure

Select one of the tabs.

SSL support on AWS

For partial SSL support on clusters running on AWS, starting with 1.3 three annotations can be added to a `LoadBalancer` service:

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-1:1
```

The first specifies the ARN of the certificate to use. It can be either a certificate from a third party issuer that was uploaded to IAM or one created within AWS Certificate Manager.

```
metadata:
  name: my-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: (https|http|ssl
```

The second annotation specifies which protocol a pod speaks. For HTTPS and SSL, the ELB will expect the pod to authenticate itself over the encrypted connection.

HTTP and HTTPS will select layer 7 proxying: the ELB will terminate the connection with the user, parse headers and inject the `X-Forwarded-For` header with the user's IP address (pods will only see the IP address of the ELB at the other end of its connection) when forwarding requests.

TCP and SSL will select layer 4 proxying: the ELB will forward traffic without modifying the headers.

In a mixed-use environment where some ports are secured and others are left unencrypted, the following annotations may be used:

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
        service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443,8443"
```

In the above example, if the service contained three ports, `80`, `443`, and `8443`, then `443` and `8443` would use the SSL certificate, but `80` would just be proxied HTTP.

PROXY protocol support on AWS

To enable PROXY protocol support for clusters running on AWS, you can use the following service annotation:

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
```

Since version 1.3.0 the use of this annotation applies to all ports proxied by the ELB and cannot be configured otherwise.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes services can be exposed on those `externalIPs`. Traffic that ingresses into the cluster with the external IP (as destination IP), on the service port, will be routed to one of the service endpoints. `externalIPs` are not managed by Kubernetes and are the responsibility of the cluster administrator.

In the ServiceSpec, `externalIPs` can be specified along with any of the `ServiceTypes`. In the example below, my-service can be accessed by clients on 80.11.12.10:80 (externalIP:port)

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  externalIPs:
  - 80.11.12.10
```

Shortcomings

Using the userspace proxy for VIPs will work at small to medium scale, but will not scale to very large clusters with thousands of Services. See the original design proposal for portals for more details.

Using the userspace proxy obscures the source-IP of a packet accessing a `Service`. This makes some kinds of firewalling impossible. The iptables proxier does not obscure in-cluster source IPs, but it does still impact clients coming through a load-balancer or node-port.

The `Type` field is designed as nested functionality - each level adds to the previous. This is not strictly required on all cloud providers (e.g. Google Compute Engine does not need to allocate a

`NodePort` to make `LoadBalancer` work, but AWS does) but the current API requires it.

Future work

In the future we envision that the proxy policy can become more nuanced than simple round robin balancing, for example master-elected or sharded. We also envision that some `Services` will have “real” load balancers, in which case the VIP will simply transport the packets there.

We intend to improve our support for L7 (HTTP) `Services`.

We intend to have more flexible ingress modes for `Services` which encompass the current `ClusterIP`, `NodePort`, and `LoadBalancer` modes and more.

The gory details of virtual IPs

The previous information should be sufficient for many people who just want to use `Services`. However, there is a lot going on behind the scenes that may be worth understanding.

Avoiding collisions

One of the primary philosophies of Kubernetes is that users should not be exposed to situations that could cause their actions to fail through no fault of their own. In this situation, we are looking at network ports - users should not have to choose a port number if that choice might collide with another user. That is an isolation failure.

In order to allow users to choose a port number for their `Services`, we must ensure that no two `Services` can collide. We do that by allocating each `Service` its own IP address.

To ensure each service receives a unique IP, an internal allocator atomically updates a global allocation map in etcd prior to creating each service. The map object must exist in the registry for services to get IPs, otherwise creations will fail with a message indicating an IP could not be allocated. A background controller is responsible for creating that map (to migrate from older versions of Kubernetes that used in memory locking) as well as checking for invalid assignments due to administrator intervention and cleaning up any IPs that were allocated but which no service currently uses.

IPs and VIPs

Unlike `Pod` IP addresses, which actually route to a fixed destination, `Service` IPs are not actually answered by a single host. Instead, we use `iptables` (packet processing logic in Linux) to define virtual IP addresses which are transparently redirected as needed. When clients connect to the VIP, their traffic is automatically transported to an appropriate endpoint. The environment variables and DNS for `Services` are actually populated in terms of the `Service`'s VIP and port.

We support two proxy modes - userspace and iptables, which operate slightly differently.

Userspace

As an example, consider the image processing application described above. When the backend `Service` is created, the Kubernetes master assigns a virtual IP address, for example 10.0.0.1. Assuming the `Service` port is 1234, the `Service` is observed by all of the `kube-proxy` instances in the cluster. When a proxy sees a new `Service`, it opens a new random port, establishes an iptables redirect from the VIP to this new port, and starts accepting connections on it.

When a client connects to the VIP the iptables rule kicks in, and redirects the packets to the `Service proxy`'s own port. The `Service proxy` chooses a backend, and starts proxying traffic from the client to the backend.

This means that `Service` owners can choose any port they want without risk of collision. Clients can simply connect to an IP and port, without being aware of which `Pods` they are actually accessing.

Iptables

Again, consider the image processing application described above. When the backend `Service` is created, the Kubernetes master assigns a virtual IP address, for example 10.0.0.1. Assuming the `Service` port is 1234, the `Service` is observed by all of the `kube-proxy` instances in the cluster. When a proxy sees a new `Service`, it installs a series of iptables rules which redirect from the VIP to per-`Service` rules. The per-`Service` rules link to per-`Endpoint` rules which redirect (Destination NAT) to the backends.

When a client connects to the VIP the iptables rule kicks in. A backend is chosen (either based on session affinity or randomly) and packets are redirected to the backend. Unlike the userspace proxy, packets are never copied to userspace, the kube-proxy does not have to be running for the VIP to work, and the client IP is not altered.

This same basic flow executes when traffic comes in through a node-port or through a load-balancer, though in those cases the client IP does get altered.

API Object

Service is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: Service API object.

For More Information

Read Connecting a Front End to a Back End Using a Service.

DNS Pods and Services

Introduction

As of Kubernetes 1.3, DNS is a built-in service launched automatically using the addon manager cluster add-on.

Kubernetes DNS schedules a DNS Pod and Service on the cluster, and configures the kubelets to tell individual containers to use the DNS Service's IP to resolve DNS names.

What things get DNS names?

Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain. This is best illustrated by example:

Assume a Service named `foo` in the Kubernetes namespace `bar`. A Pod running in namespace `bar` can look up this service by simply doing a DNS query for `foo`. A Pod running in namespace `quux` can look up this service by doing a DNS query for `foo.bar`.

Supported DNS schema

The following sections detail the supported record types and layout that is supported. Any other layout or names or queries that happen to work are considered implementation details and are subject to change without warning.

Services

A records

“Normal” (not headless) Services are assigned a DNS A record for a name of the form `my-svc.my-namespace.svc.cluster.local`. This resolves to the cluster IP of the Service.

“Headless” (without a cluster IP) Services are also assigned a DNS A record for a name of the form `my-svc.my-namespace.svc.cluster.local`. Unlike normal Services, this resolves to the set of IPs of the pods selected by the Service. Clients are expected to consume the set or else use standard round-robin selection from the set.

SRV records

SRV Records are created for named ports that are part of normal or Headless Services. For each named port, the SRV record would have the form `_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster.local`. For a regular service, this resolves to the port number and the CNAME: `my-svc.my-namespace.svc.cluster.local`. For a headless service, this resolves to multiple answers, one for each pod that is backing the service, and contains the port number and a CNAME of the pod of the form `auto-generated-name.my-svc.my-namespace.svc.cluster.local`.

Backwards compatibility

Previous versions of kube-dns made names of the form `my-svc.my-namespace.cluster.local` (the ‘svc’ level was added later). This is no longer supported.

Pods

A Records

When enabled, pods are assigned a DNS A record in the form of `pod-ip-address.my-namespace.pod.cluster.local`.

For example, a pod with IP `1.2.3.4` in the namespace `default` with a DNS name of `cluster.local` would have an entry: `1-2-3-4.default.pod.cluster.local`.

A Records and hostname based on Pod’s hostname and subdomain fields

Currently when a pod is created, its hostname is the Pod’s `metadata.name` value.

With v1.2, users can specify a Pod annotation, `pod.beta.kubernetes.io/hostname`, to specify what the Pod's hostname should be. The Pod annotation, if specified, takes precedence over the Pod's name, to be the hostname of the pod. For example, given a Pod with annotation `pod.beta.kubernetes.io/hostname: my-pod-name`, the Pod will have its hostname set to "my-pod-name".

With v1.3, the PodSpec has a `hostname` field, which can be used to specify the Pod's hostname. This field value takes precedence over the `pod.beta.kubernetes.io/hostname` annotation value.

v1.2 introduces a beta feature where the user can specify a Pod annotation, `pod.beta.kubernetes.io/subdomain`, to specify the Pod's subdomain. The final domain will be " ...svc.". For example, a Pod with the hostname annotation set to "foo", and the subdomain annotation set to "bar", in namespace "my-namespace", will have the FQDN "foo.bar.my-namespace.svc.cluster.local"

With v1.3, the PodSpec has a `subdomain` field, which can be used to specify the Pod's subdomain. This field value takes precedence over the `pod.beta.kubernetes.io/subdomain` annotation value.

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
  - name: foo # Actually, no port is needed.
    port: 1234
    targetPort: 1234
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox1
  labels:
    name: busybox
spec:
  hostname: busybox-1
  subdomain: default-subdomain
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
---
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox
spec:
  hostname: busybox-2
  subdomain: default-subdomain
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

If there exists a headless service in the same namespace as the pod and with the same name as the subdomain, the cluster's KubeDNS Server also returns an A record for the Pod's fully qualified hostname. Given a Pod with the hostname set to "busybox-1" and the subdomain set to "default-subdomain”, and a headless Service named “default-subdomain” in the same namespace, the pod will see its own FQDN as “busybox-1.default-subdomain.my-namespace.svc.cluster.local”. DNS serves an A record at that name, pointing to the Pod’s IP. Both pods “busybox1” and “busybox2” can have their distinct A records.

As of Kubernetes v1.2, the Endpoints object also has the annotation `endpoints.beta.kubernetes.io/hostnames-map`. Its value is the json representation of map[string(IP)][endpoints.HostRecord], for example: ‘{“10.245.1.6”:{HostName: “my-webserver”}}’. If the Endpoints are for a headless service, an A record is created with the format ...svc. For the example json, if endpoints are for a headless service named "bar", and one of the endpoints has IP "10.245.1.6", an A record is created with the name "my-webserver.bar.my-namespace.svc.cluster.local" and the A record lookup would return "10.245.1.6". This endpoints annotation generally does not need to be specified by end-users, but can used by the internal service controller to deliver the aforementioned feature.

With v1.3, The Endpoints object can specify the `hostname` for any endpoint, along with its IP. The hostname field takes precedence over the hostname value that might have been specified via the `endpoints.beta.kubernetes.io/hostnames-map` annotation.

With v1.3, the following annotations are deprecated: `pod.beta.kubernetes.io/hostname`, `pod.beta.kubernetes.io/subdomain`, `endpoints.beta.kubernetes.io/hostnames-map`.

How do I test if it is working?

Create a simple Pod to use as a test environment

Create a file named busybox.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

Then create a pod using this file:

```
kubectl create -f busybox.yaml
```

Wait for this pod to go into the running state

You can get its status with: `kubectl get pods busybox`

You should see:

```
NAME      READY     STATUS    RESTARTS   AGE
busybox   1/1       Running   0          <some-time>
```

Validate that DNS is working

Once that pod is running, you can exec nslookup in that environment:

```
kubectl exec -ti busybox -- nslookup kubernetes.default
```

You should see something like:

```
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes.default
Address 1: 10.0.0.1
```

If you see that, DNS is working correctly.

Troubleshooting Tips

If the nslookup command fails, check the following:

Check the local DNS configuration first

Take a look inside the resolv.conf file. (See “Inheriting DNS from the node” and “Known issues” below for more information)

```
kubectl exec busybox cat /etc/resolv.conf
```

Verify that the search path and name server are set up like the following (note that search path may vary for different cloud providers):

```
search default.svc.cluster.local svc.cluster.local cluster.local google.internal c
nameserver 10.0.0.10
options ndots:5
```

DNS Policy

By default, DNS policy for a pod is ‘ClusterFirst’. So pods running with hostNetwork cannot resolve DNS names. To have DNS options set along with hostNetwork, you should specify DNS policy explicitly to ‘ClusterFirstWithHostNet’. Update the busybox.yaml as following:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Quick diagnosis

Errors such as the following indicate a problem with the kube-dns add-on or associated Services:

```
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10

nslookup: can't resolve 'kubernetes.default'
```

or

```
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

nslookup: can't resolve 'kubernetes.default'
```

Check if the DNS pod is running

Use the kubectl get pods command to verify that the DNS pod is running.

```
kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
```

You should see something like:

```
NAME                                                        READY     STATUS    RES
...
kube-dns-v19-ezo1y                                          3/3       Running   0
...
```

If you see that no pod is running or that the pod has failed/completed, the DNS add-on may not be deployed by default in your current environment and you will have to deploy it manually.

Check for Errors in the DNS pod

Use `kubectl logs` command to see logs for the DNS daemons.

```
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l
```

See if there is any suspicious log. W, E, F letter at the beginning represent Warning, Error and Failure. Please search for entries that have these as the logging level and use kubernetes issues to report unexpected errors.

Is DNS service up?

Verify that the DNS service is up by using the `kubectl get service` command.

```
kubectl get svc --namespace=kube-system
```

You should see:

```
NAME                    CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
...
kube-dns                10.0.0.10      <none>        53/UDP,53/TCP        1h
...
```

If you have created the service or in the case it should be created by default but it does not appear, see this debugging services page for more information.

Are DNS endpoints exposed?

You can verify that DNS endpoints are exposed by using the `kubectl get endpoints` command.

```
kubectl get ep kube-dns --namespace=kube-system
```

You should see something like:
`NAME ENDPOINTS AGE kube-dns 10.180.3.17:53,10.180.3.17:53 1h`

If you do not see the endpoints, see endpoints section in the debugging services documentation.

For additional Kubernetes DNS examples, see the cluster-dns examples in the Kubernetes GitHub repository.

Kubernetes Federation (Multiple Zone support)

Release 1.3 introduced Cluster Federation support for multi-site Kubernetes installations. This required some minor (backward-compatible) changes to the way the Kubernetes cluster DNS server processes DNS queries, to facilitate the lookup of federated services (which span multiple Kubernetes clusters). See the Cluster Federation Administrators' Guide for more details on Cluster Federation and multi-site support.

How it Works

The running Kubernetes DNS pod holds 3 containers - kubedns, dnsmasq and a health check called healthz. The kubedns process watches the Kubernetes master for changes in Services and Endpoints, and maintains in-memory lookup structures to service DNS requests. The dnsmasq container adds DNS caching to improve performance. The healthz container provides a single health check endpoint while performing dual healthchecks (for dnsmasq and kubedns).

The DNS pod is exposed as a Kubernetes Service with a static IP. Once assigned the kubelet passes DNS configured using the `--cluster-dns=10.0.0.10` flag to each container.

DNS names also need domains. The local domain is configurable, in the kubelet using the flag `--cluster-domain=<default local domain>` .

The Kubernetes cluster DNS server (based off the SkyDNS library) supports forward lookups (A records), service lookups (SRV records) and reverse IP address lookups (PTR records).

Inheriting DNS from the node

When running a pod, kubelet will prepend the cluster DNS server and search paths to the node's own DNS settings. If the node is able to resolve DNS names specific to the larger environment, pods should be able to, also. See "Known issues" below for a caveat.

If you don't want this, or if you want a different DNS config for pods, you can use the kubelet's `--resolv-conf` flag. Setting it to "" means that pods will not inherit DNS. Setting it to a valid file path means that kubelet will use this file instead of `/etc/resolv.conf` for DNS inheritance.

Known issues

Kubernetes installs do not configure the nodes' resolv.conf files to use the cluster DNS by default, because that process is inherently distro-specific. This should probably be implemented eventually.

Linux's libc is impossibly stuck (see this bug from 2005) with limits of just 3 DNS `nameserver` records and 6 DNS `search` records. Kubernetes needs to consume 1 `nameserver` record and 3 `search` records. This means that if a local installation already uses 3 `nameserver`s or uses more than 3 `search`es, some of those settings will be lost. As a partial workaround, the node can run `dnsmasq` which will provide more `nameserver` entries, but not more `search` entries. You can also use kubelet's `--resolv-conf` flag.

If you are using Alpine version 3.3 or earlier as your base image, DNS may not work properly owing to a known issue with Alpine. Check here for more information.

References

- Docs for the DNS cluster addon

What's next

- Autoscaling the DNS Service in a Cluster.

Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network. Before discussing the Kubernetes approach to networking, it is worthwhile to contrast it with the "normal" way networking works with Docker.

By default, Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, they must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or else be allocated ports dynamically.

Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control. Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. We give every pod its own cluster-private-IP address so you do not need to explicitly create links between pods or mapping container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document will elaborate on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept. The same principles are embodied in a more complete Jenkins CI application.

Exposing pods to the cluster

We did this in a previous example, but let's do it once again and focus on the networking perspective. Create an nginx pod, and note that it has a container port specification:

run-my-nginx.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

This makes it accessible from any node in your cluster. Check the nodes the pod is running on:

```
$ kubectl create -f ./run-my-nginx.yaml
$ kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE       IP
my-nginx-3800858182-jr4a2   1/1       Running   0          13s       10.244.3.4
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5
```

Check your pods' IPs:

```
$ kubectl get pods -l run=my-nginx -o yaml | grep podIP
    podIP: 10.244.3.4
    podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are *not* using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node's interfaces, but the need for this is radically diminished because of the networking model.

You can read more about how we achieve this if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

You can create a Service for your 2 nginx replicas with `kubectl expose`:

```
$ kubectl expose deployment/my-nginx
service "my-nginx" exposed
```

This is equivalent to `kubectl create -f` the following yaml:

nginx-svc.yaml

nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Service port (`targetPort` : is the port the container accepts traffic on, `port` : is the abstracted Service port, which can be any port other pods use to access the Service). View service API object to see the list of supported fields in service definition. Check your Service:

```
$ kubectl get svc my-nginx
NAME       CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-nginx   10.0.162.149   <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of pods. These pods are exposed through `endpoints` . The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named `my-nginx` . When a pod dies, it is automatically removed from the endpoints, and new pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the pods created in the first step:

```
$ kubectl describe svc my-nginx
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
Annotations:         <none>
Selector:            run=my-nginx
Type:                ClusterIP
IP:                  10.0.162.149
Port:                <unset> 80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
Session Affinity:    None
Events:              <none>

$ kubectl get ep my-nginx
NAME       ENDPOINTS                     AGE
my-nginx   10.244.2.5:80,10.244.3.4:80   1m
```

You should now be able to curl the nginx Service on `<CLUSTER-IP>:<PORT>` from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire, if you're curious about how this works you can read more about the service proxy.

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the kube-dns cluster addon.

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx pods (your pod name will be different):

```
$ kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 pods and waiting for the Deployment to recreate them. This time around the Service exists *before* the replicas. This will give you scheduler-level Service spreading of your pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
$ kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-ngin

$ kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE     IP            N
my-nginx-3800858182-e9ihh   1/1       Running   0          5s      10.244.2.7    k
my-nginx-3800858182-j4rm4   1/1       Running   0          5s      10.244.3.8    k
```

You may notice that the pods have different names, since they are killed and recreated.

```
$ kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that uses skydns to automatically assign dns names to other Services. You can check if it's running on your cluster:

```
$ kubectl get services kube-dns --namespace=kube-system
NAME       CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

If it isn't running, you can enable it. The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a dns server that has assigned a name to that IP (the kube-dns cluster addon), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname). Let's run another curl application to test this:

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty
Waiting for pod default/curl-131556218-9fnch to be running, status is Pending, pod
Hit enter for command prompt
```

Then, hit enter and run `nslookup my-nginx`:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

You can acquire all these from the nginx https example, in short:

```
$ make keys secret KEY=/tmp/nginx.key CERT=/tmp/nginx.crt SECRET=/tmp/secret.json
$ kubectl create -f /tmp/secret.json
secret "nginxsecret" created
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           Opaque                                2         1m
```

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

nginx-secure-app.yaml

nginx-secure-app.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves http traffic on port 80 and https traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at /etc/nginx/ssl. This is setup *before* the nginx server is started.

```
$ kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.ya
```

At this point you can reach the nginx server from any node.

```
$ kubectl get pods -o yaml | grep -i podip
    podIP: 10.244.3.5
node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the `-k` parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

curlpod.yaml

curlpod.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

```
$ kubectl create -f ./curlpod.yaml
$ kubectl get pods -l app=curlpod
NAME                               READY     STATUS    RESTARTS   AGE
curl-deployment-1515033274-1410r   1/1       Running   0          1m
$ kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used `NodePort`, so your nginx https replica is ready to serve traffic on the internet if your node has a public IP.

```
$ kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
    run: my-nginx

$ kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer, just change the `Type` of `my-nginx` Service from `NodePort` to `LoadBalancer`:

```
$ kubectl edit svc my-nginx
$ kubectl get svc my-nginx
NAME       CLUSTER-IP     EXTERNAL-IP        PORT(S)                AGE
my-nginx   10.0.162.149   162.222.184.144    80/TCP,81/TCP,82/TCP   21s

$ curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
```

The IP address in the `EXTERNAL-IP` column is the one that is available on the public internet. The `CLUSTER-IP` is only available inside your cluster/private cloud network.

Note that on AWS, type `LoadBalancer` creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard `kubectl get svc` output, in fact, so you'll need to do `kubectl describe service my-nginx` to see it. You'll see something like this:

```
$ kubectl describe service my-nginx
...
LoadBalancer Ingress:   a320587ffd19711e5a37606cf4a74574-1142138393.us-east-1.elb.
...
```

Further reading

Kubernetes also supports Federated Services, which can span multiple clusters and cloud providers, to provide increased availability, better fault tolerance and greater scalability for your services. See the Federated Services User Guide for further information.

What's next?

Learn about more Kubernetes features that will help you run containers reliably in production.

Ingress Resources

Terminology

Throughout this doc you will see a few terms that are sometimes used interchangeably elsewhere, that might cause confusion. This section attempts to clarify them.

- Node: A single virtual or physical machine in a Kubernetes cluster.
- Cluster: A group of nodes firewalled from the internet, that are the primary compute resources managed by Kubernetes.
- Edge router: A router that enforces the firewall policy for your cluster. This could be a gateway managed by a cloud provider or a physical piece of hardware.
- Cluster network: A set of links, logical or physical, that facilitate communication within a cluster according to the Kubernetes networking model. Examples of a Cluster network include Overlays such as flannel or SDNs such as OVS.
- Service: A Kubernetes Service that identifies a set of pods using label selectors. Unless mentioned otherwise, Services are assumed to have virtual IPs only routable within the cluster network.

What is Ingress?

Typically, services and pods have IPs only routable by the cluster network. All traffic that ends up at an edge router is either dropped or forwarded elsewhere. Conceptually, this might look like:

```
  internet
      |
------------
[ Services ]
```

An Ingress is a collection of rules that allow inbound connections to reach the cluster services.

```
   internet
      |
 [ Ingress ]
 --|-----|--
 [ Services ]
```

It can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting etc. Users request ingress by POSTing the Ingress resource to the API server. An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic in an HA manner.

Prerequisites

Before you start using the Ingress resource, there are a few things you should understand. The Ingress is a beta resource, not available in any Kubernetes release prior to 1.1. You need an Ingress controller to satisfy an Ingress, simply creating the resource will have no effect.

GCE/GKE deploys an ingress controller on the master. You can deploy any number of custom ingress controllers in a pod. You must annotate each ingress with the appropriate class, as indicated here and here.

Make sure you review the beta limitations of this controller. In environments other than GCE/GKE, you need to deploy a controller as a pod.

The Ingress Resource

A minimal Ingress might look like:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

POSTing this to the API server will have no effect if you have not configured an Ingress controller.

Lines 1-6: As with all other Kubernetes config, an Ingress needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see deploying applications, configuring containers, managing resources and ingress configuration rewrite.

Lines 7-9: Ingress spec has all the information needed to configure a loadbalancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Currently the Ingress resource only supports http rules.

Lines 10-11: Each http rule contains the following information: A host (e.g.: foo.bar.com, defaults to * in this example), a list of paths (e.g.: /testpath) each of which has an associated backend (test:80). Both the host and path must match the content of an incoming request before the loadbalancer directs traffic to the backend.

Lines 12-14: A backend is a service:port combination as described in the services doc. Ingress traffic is typically sent directly to the endpoints matching a backend.

Global Parameters: For the sake of simplicity the example Ingress has no global parameters, see the API reference for a full definition of the resource. One can specify a global default backend in the absence of which requests that don't match a path in the spec are sent to the default backend of the Ingress controller.

Ingress controllers

In order for the Ingress resource to work, the cluster must have an Ingress controller running. This is unlike other types of controllers, which typically run as part of the `kube-controller-manager` binary, and which are typically started automatically as part of cluster creation. You need to choose the ingress controller implementation that is the best fit for your cluster, or implement one. Examples and instructions can be found here.

Before you begin

The following document describes a set of cross platform features exposed through the Ingress resource. Ideally, all Ingress controllers should fulfill this specification, but we're not there yet. The docs for the GCE and nginx controllers are here and here respectively. **Make sure you review controller specific docs so you understand the caveats of each one**.

Types of Ingress

Single Service Ingress

There are existing Kubernetes concepts that allow you to expose a single service (see alternatives), however you can do so through an Ingress as well, by specifying a *default backend* with no rules.

ingress.yaml

ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

If you create it using `kubectl create -f` you should see:

```
$ kubectl get ing
NAME                RULE          BACKEND        ADDRESS
test-ingress        -             testsvc:80     107.178.254.228
```

Where `107.178.254.228` is the IP allocated by the Ingress controller to satisfy this Ingress. The `RULE` column shows that all traffic sent to the IP is directed to the Kubernetes Service listed under `BACKEND`.

Simple fanout

As described previously, pods within kubernetes have IPs only visible on the cluster network, so we need something at the edge accepting ingress traffic and proxying it to the right endpoints. This component is usually a highly available loadbalancer. An Ingress allows you to keep the number of loadbalancers down to a minimum, for example, a setup like:

```
foo.bar.com -> 178.91.123.132 -> / foo    s1:80
                                 / bar    s2:80
```

would require an Ingress such as:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

When you create the Ingress with `kubectl create -f`:

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -
          foo.bar.com
          /foo          s1:80
          /bar          s2:80
```

The Ingress controller will provision an implementation specific loadbalancer that satisfies the Ingress, as long as the services (s1, s2) exist. When it has done so, you will see the address of the loadbalancer under the last column of the Ingress.

Name based virtual hosting

Name-based virtual hosts use multiple host names for the same IP address.

```
foo.bar.com --|                 |-> foo.bar.com s1:80
              | 178.91.123.132  |
bar.foo.com --|                 |-> bar.foo.com s2:80
```

The following Ingress tells the backing loadbalancer to route requests based on the Host header.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

Default Backends: An Ingress with no rules, like the one shown in the previous section, sends all traffic to a single default backend. You can use the same technique to tell a loadbalancer where to find your website's 404 page, by specifying a set of rules *and* a default backend. Traffic is routed to your default backend if none of the Hosts in your Ingress match the Host in the request header, and/or none of the paths match the URL of the request.

TLS

You can secure an Ingress by specifying a secret that contains a TLS private key and certificate. Currently the Ingress only supports a single TLS port, 443, and assumes TLS termination. If the TLS configuration section in an Ingress specifies different hosts, they will be multiplexed on the same port according to the hostname specified through the SNI TLS extension (provided the Ingress controller supports SNI). The TLS secret must contain keys named `tls.crt` and `tls.key` that contain the certificate and private key to use for TLS, e.g.:

```yaml
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret
  namespace: default
type: Opaque
```

Referencing this secret in an Ingress will tell the Ingress controller to secure the channel from the client to the loadbalancer using TLS:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
    - secretName: testsecret
  backend:
    serviceName: s1
    servicePort: 80
```

Note that there is a gap between TLS features supported by various Ingress controllers. Please refer to documentation on nginx, GCE, or any other platform specific Ingress controller to understand how TLS works in your environment.

Loadbalancing

An Ingress controller is bootstrapped with some loadbalancing policy settings that it applies to all Ingress, such as the loadbalancing algorithm, backend weight scheme etc. More advanced loadbalancing concepts (e.g.: persistent sessions, dynamic weights) are not yet exposed through the Ingress. You can still get these features through the service loadbalancer. With time, we plan to distill loadbalancing patterns that are applicable cross platform into the Ingress resource.

It's also worth noting that even though health checks are not exposed directly through the Ingress, there exist parallel concepts in Kubernetes such as readiness probes which allow you to achieve the same end result. Please review the controller specific docs to see how they handle health checks (nginx, GCE).

Updating an Ingress

Say you'd like to add a new Host to an existing Ingress, you can update it by editing the resource:

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
$ kubectl edit ing test
```

This should pop up an editor with the existing yaml, modify it to include the new Host:

```
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
        path: /foo
  - host: bar.baz.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
        path: /foo
..
```

Saving it will update the resource in the API server, which should tell the Ingress controller to reconfigure the loadbalancer.

```
$ kubectl get ing
NAME      RULE          BACKEND   ADDRESS
test      -                       178.91.123.132
          foo.bar.com
          /foo          s1:80
          bar.baz.com
          /foo          s2:80
```

You can achieve the same by invoking `kubectl replace -f` on a modified Ingress yaml file.

Failing across availability zones

Techniques for spreading traffic across failure domains differs between cloud providers. Please check the documentation of the relevant Ingress controller for details. Please refer to the federation doc for details on deploying Ingress in a federated cluster.

Future Work

- Various modes of HTTPS/TLS support (e.g.: SNI, re-encryption)
- Requesting an IP or Hostname via claims
- Combining L4 and L7 Ingress
- More Ingress controllers

Please track the L7 and Ingress proposal for more details on the evolution of the resource, and the Ingress repository for more details on the evolution of various Ingress controllers.

Alternatives

You can expose a Service in multiple ways that don't directly involve the Ingress resource:

- Use Service.Type=LoadBalancer
- Use Service.Type=NodePort
- Use a Port Proxy
- Deploy the Service loadbalancer. This allows you to share a single IP among multiple Services and achieve more advanced loadbalancing through Service Annotations.

Network Policies

A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

`NetworkPolicy` resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.

Prerequisites

Network policies are implemented by the network plugin, so you must be using a networking solution which supports `NetworkPolicy` - simply creating the resource without a controller to implement it will have no effect.

Isolated and Non-isolated Pods

By default, pods are non-isolated; they accept traffic from any source.

Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

The NetworkPolicy Resource

See the api-reference for a full definition of the resource.

An example `NetworkPolicy` might look like this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

POSTing this to the API server will have no effect unless your chosen networking solution supports network policy.

Mandatory Fields: As with all other Kubernetes config, a `NetworkPolicy` needs `apiVersion` , `kind` , and `metadata` fields. For general information about working with config files, see here, here, and here.

spec: `NetworkPolicy` spec has all the information needed to define a particular network policy in the given namespace.

podSelector: Each `NetworkPolicy` includes a `podSelector` which selects the grouping of pods to which the policy applies. Since `NetworkPolicy` currently only supports defining `ingress` rules, this `podSelector` essentially defines the "destination pods" for the policy. The example policy selects pods with the label "role=db". An empty `podSelector` selects all pods in the namespace.

policyTypes: Each `NetworkPolicy` includes a `policyTypes` list which may include either `Ingress` , `Egress` , or both. The `policyTypes` field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no `policyTypes` are specified on a NetworkPolicy then by default `Ingress` will always be set and `Egress` will be set if the NetworkPolicy has any egress rules.

ingress: Each `NetworkPolicy` may include a list of whitelist `ingress` rules. Each rule allows traffic which matches both the `from` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port, from either of two sources, the first specified via a `namespaceSelector` and the second specified via a `podSelector` .

egress: Each `NetworkPolicy` may include a list of whitelist `egress` rules. Each rule allows traffic which matches both the `to` and `ports` sections. The example policy contains a single rule, which matches traffic on a single port to any destination in `10.0.0.0/24` .

ipBlock: `ipBlock` describes a particular CIDR that is allowed to the pods matched by a NetworkPolicySpec's podSelector. The `except` entry is a slice of CIDRs that should not be included within an IP Block. Except values will be rejected if they are outside the CIDR range.

So, the example NetworkPolicy:

1. isolates "role=db" pods in the "default" namespace for both ingress and egress traffic (if they weren't already isolated)

2. allows connections to TCP port 6379 of “role=db” pods in the “default” namespace from any pod in the “default” namespace with the label “role=frontend”
3. allows connections to TCP port 6379 of “role=db” pods in the “default” namespace from any pod in a namespace with the label “project=myproject”
4. allows connections from any pod in the “default” namespace with the label “role=db” to CIDR 10.0.0.0/24 on TCP port 5978

See the NetworkPolicy getting started guide for further examples.

Default policies

By default, if no policies exist in a namespace, then all ingress and egress traffic is allowed to and from pods in that namespace. The following examples let you change the default behavior in that namespace.

Default deny all ingress traffic

You can create a “default” isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any ingress traffic to those pods.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
  policyTypes:
  - Ingress
```

This ensures that even pods that aren’t selected by any other NetworkPolicy will still be isolated. This policy does not change the default egress isolation behavior.

Default allow all ingress traffic

If you want to allow all traffic to all pods in a namespace (even if policies are added that cause some pods to be treated as “isolated”), you can create a policy that explicitly allows all traffic in that namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
  ingress:
  - {}
```

Default deny all egress traffic.

You can create a “default” egress isolation policy for a namespace by creating a NetworkPolicy that selects all pods but does not allow any egress traffic from those pods.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
  policyTypes:
  - Egress
```

This ensures that even pods that aren’t selected by any other NetworkPolicy will not be allowed egress traffic. This policy does not change the default ingress isolation behavior.

Default allow all egress traffic

If you want to allow all traffic from all pods in a namespace (even if policies are added that cause some pods to be treated as “isolated”), you can create a policy that explicitly allows all egress traffic in that namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector:
  egress:
  - {}
```

Default deny all ingress and all egress traffic

You can create a “default” policy for a namespace which prevents all ingress AND egress traffic by creating the following NetworkPolicy in that namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
  policyTypes:
  - Ingress
  - Egress
```

This ensures that even pods that aren’t selected by any other NetworkPolicy will not be allowed ingress or egress traffic.

What’s next?

- See the Declare Network Policy walkthrough for further examples.

Adding entries to Pod /etc/hosts with HostAliases

Adding entries to a Pod's /etc/hosts file provides Pod-level override of hostname resolution when DNS and other options are not applicable. In 1.7, users can add these custom entries with the HostAliases field in PodSpec.

Modification not using HostAliases is not suggested because the file is managed by Kubelet and can be overwritten on during Pod creation/restart.

Default Hosts File Content

Lets start an Nginx Pod which is assigned an Pod IP:

```
$ kubectl run nginx --image nginx --generator=run-pod/v1
pod "nginx" created

$ kubectl get pods --output=wide
NAME     READY     STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1       Running   0          13s    10.200.0.4   worker0
```

The hosts file content would look like this:

```
$ kubectl exec nginx -- cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.4      nginx
```

by default, the hosts file only includes ipv4 and ipv6 boilerplates like `localhost` and its own hostname.

Adding Additional Entries with HostAliases

In addition to the default boilerplate, we can add additional entries to the hosts file to resolve `foo.local`, `bar.local` to `127.0.0.1` and `foo.remote`, `bar.remote` to `10.1.2.3`, we can by adding HostAliases to the Pod under `.spec.hostAliases`:

hostaliases-pod.yaml

hostaliases-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
    args:
    - "/etc/hosts"
```

This Pod can be started with the following commands:

```
$ kubectl apply -f hostaliases-pod.yaml
pod "hostaliases-pod" created

$ kubectl get pod -a -o=wide
NAME                           READY     STATUS      RESTARTS   AGE       IP
hostaliases-pod                0/1       Completed   0          6s        10.244.1
```

The hosts file content would look like this:

```
$ kubectl logs hostaliases-pod
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.244.135.10   hostaliases-pod
127.0.0.1       foo.local
127.0.0.1       bar.local
10.1.2.3        foo.remote
10.1.2.3        bar.remote
```

With the additional entries specified at the bottom.

Limitations

HostAlias is only supported in 1.7+.

HostAlias support in 1.7 is limited to non-hostNetwork Pods because kubelet only manages the hosts file for non-hostNetwork Pods.

In 1.8, HostAlias is supported for all Pods regardless of network configuration.

Why Does Kubelet Manage the Hosts File?

Kubelet manages the hosts file for each container of the Pod to prevent Docker from modifying the file after the containers have already been started.

Because of the managed-nature of the file, any user-written content will be overwritten whenever the hosts file is remounted by Kubelet in the event of a container restart or a Pod reschedule. Thus, it is not suggested to modify the contents of the file.

Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. First, when a container crashes, kubelet will restart it, but the files will be lost - the container starts with a clean state. Second, when running containers together in a `Pod` it is often necessary to share files between those containers. The Kubernetes `Volume` abstraction solves both of these problems.

Familiarity with pods is suggested.

Background

Docker also has a concept of volumes, though it is somewhat looser and less managed. In Docker, a volume is simply a directory on disk or in another container. Lifetimes are not managed and until very recently there were only local-disk-backed volumes. Docker now provides volume drivers, but the functionality is very limited for now (e.g. as of Docker 1.7 only one volume driver is allowed per container and there is no way to pass parameters to volumes).

A Kubernetes volume, on the other hand, has an explicit lifetime - the same as the pod that encloses it. Consequently, a volume outlives any containers that run within the Pod, and data is preserved across Container restarts. Of course, when a Pod ceases to exist, the volume will cease to exist, too. Perhaps more importantly than this, Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously.

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

To use a volume, a pod specifies what volumes to provide for the pod (the `spec.volumes` field) and where to mount those into containers(the `spec.containers.volumeMounts` field).

A process in a container sees a filesystem view composed from their Docker image and volumes. The Docker image is at the root of the filesystem hierarchy, and any volumes are mounted at the specified paths within the image. Volumes can not mount onto other volumes or have hard links to other volumes. Each container in the Pod must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

- `emptyDir`
- `hostPath`
- `gcePersistentDisk`
- `awsElasticBlockStore`
- `nfs`
- `iscsi`
- `fc (fibre channel)`
- `flocker`
- `glusterfs`
- `rbd`
- `cephfs`
- `gitRepo`
- `secret`
- `persistentVolumeClaim`
- `downwardAPI`
- `projected`

- `azureFileVolume`
- `azureDisk`
- `vsphereVolume`
- `Quobyte`
- `PortworxVolume`
- `ScaleIO`
- `StorageOS`
- `local`

We welcome additional contributions.

emptyDir

An `emptyDir` volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node. As the name says, it is initially empty. Containers in the pod can all read and write the same files in the `emptyDir` volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the `emptyDir` is deleted forever.

> **Note:** a container crashing does *NOT* remove a pod from a node, so the data in an `emptyDir` volume is safe across container crashes.

Some uses for an `emptyDir` are:

- scratch space, such as for a disk-based merge sort
- checkpointing a long computation for recovery from crashes
- holding files that a content-manager container fetches while a webserver container serves the data

By default, `emptyDir` volumes are stored on whatever medium is backing the node - that might be disk or SSD or network storage, depending on your environment. However, you can set the `emptyDir.medium` field to `"Memory"` to tell Kubernetes to mount a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that unlike disks, tmpfs is cleared on node reboot and any files you write will count against your container's memory limit.

Example pod

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath

A `hostPath` volume mounts a file or directory from the host node's filesystem into your pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

For example, some uses for a `hostPath` are:

- running a container that needs access to Docker internals; use a `hostPath` of `/var/lib/docker`
- running cAdvisor in a container; use a `hostPath` of `/dev/cgroups`
- allowing a pod to specify whether a given hostPath should exist prior to the pod running, whether it should be created, and what it should exist as

In addition to the required `path` property, user can optionally specify a `type` for a `hostPath` volume.

The supported values for field `type` are:

Value	Behavior
	Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the hostPath volume.
`DirectoryOrCreate`	If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet.
`Directory`	A directory must exist at the given path
`FileOrCreate`	If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet.
`File`	A file must exist at the given path
`Socket`	A UNIX socket must exist at the given path
`CharDevice`	A character device must exist at the given path
`BlockDevice`	A block device must exist at the given path

Watch out when using this type of volume, because:

- pods with identical configuration (such as created from a podTemplate) may behave differently on different nodes due to different files on the nodes
- when Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a `hostPath`
- the files or directories created on the underlying hosts are only writable by root. You either need to run your process as root in a privileged container or modify the file permissions on the host to be able to write to a `hostPath` volume

Example pod

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
      # this field is optional
      type: Directory
```

gcePersistentDisk

A `gcePersistentDisk` volume mounts a Google Compute Engine (GCE) Persistent Disk into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a PD are preserved and the volume is merely unmounted. This means that a PD can be pre-populated with data, and that data can be "handed off" between pods.

> **Important:** You must create a PD using `gcloud` or the GCE API or UI before you can use it.

There are some restrictions when using a `gcePersistentDisk`:

- the nodes on which pods are running must be GCE VMs
- those VMs need to be in the same GCE project and zone as the PD

A feature of PD is that they can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a PD with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, PDs can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

Using a PD on a pod controlled by a ReplicationController will fail unless the PD is read-only or the replica count is 0 or 1.

Creating a PD

Before you can use a GCE PD with a pod, you need to create it.

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

Example pod

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

awsElasticBlockStore

An `awsElasticBlockStore` volume mounts an Amazon Web Services (AWS) EBS Volume into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted. This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between pods.

> **Important:** You must create an EBS volume using `aws ec2 create-volume` or the AWS API before you can use it.

There are some restrictions when using an awsElasticBlockStore volume:

- the nodes on which pods are running must be AWS EC2 instances
- those instances need to be in the same region and availability-zone as the EBS volume
- EBS only supports a single EC2 instance mounting a volume

Creating an EBS volume

Before you can use an EBS volume with a pod, you need to create it.

```
aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2
```

Make sure the zone matches the zone you brought up your cluster in. (And also check that the size and EBS volume type are suitable for your use!)

AWS EBS Example configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

nfs

An `nfs` volume allows an existing NFS (Network File System) share to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `nfs` volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data, and that data can be "handed off" between pods. NFS can be mounted by multiple writers simultaneously.

> **Important:** You must have your own NFS server running with the share exported before you can use it.

See the NFS example for more details.

iscsi

An `iscsi` volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of an `iscsi` volume are preserved and the volume is merely unmounted. This means that an iscsi volume can be pre-populated with data, and that data can be "handed off" between pods.

> **Important:** You must have your own iSCSI server running with the volume created before you can use it.

A feature of iSCSI is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

See the iSCSI example for more details.

fc (fibre channel)

An `fc` volume allows an existing fibre channel volume to be mounted in a pod. You can specify single or multiple target World Wide Names using the parameter `targetWWNs` in your volume configuration. If multiple WWNs are specified, targetWWNs expect that those WWNs are from multi-path connections.

> **Important:** You must configure FC SAN Zoning to allocate and mask those LUNs (volumes) to the target WWNs beforehand so that Kubernetes hosts can access them.

See the FC example for more details.

flocker

Flocker is an open-source clustered container data volume manager. It provides management and orchestration of data volumes backed by a variety of storage backends.

A `flocker` volume allows a Flocker dataset to be mounted into a pod. If the dataset does not already exist in Flocker, it needs to be first created with the Flocker CLI or by using the Flocker API. If the dataset already exists it will be reattached by Flocker to the node that the pod is scheduled. This means data can be "handed off" between pods as required.

> **Important:** You must have your own Flocker installation running before you can use it.

See the Flocker example for more details.

glusterfs

A `glusterfs` volume allows a Glusterfs (an open source networked filesystem) volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `glusterfs` volume are preserved and the volume is merely unmounted. This means that a glusterfs volume can be pre-populated with data, and that data can be "handed off" between pods. GlusterFS can be mounted by multiple writers simultaneously.

> **Important:** You must have your own GlusterFS installation running before you can use it.

See the GlusterFS example for more details.

rbd

An `rbd` volume allows a Rados Block Device volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `rbd` volume are preserved and the volume is merely unmounted. This means that a RBD volume can be pre-populated with data, and that data can be "handed off" between pods.

> **Important:** You must have your own Ceph installation running before you can use RBD.

A feature of RBD is that it can be mounted as read-only by multiple consumers simultaneously. This means that you can pre-populate a volume with your dataset and then serve it in parallel from as many pods as you need. Unfortunately, RBD volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

See the RBD example for more details.

cephfs

A `cephfs` volume allows an existing CephFS volume to be mounted into your pod. Unlike `emptyDir`, which is erased when a Pod is removed, the contents of a `cephfs` volume are preserved and the volume is merely unmounted. This means that a CephFS volume can be pre-populated with data, and that data can be "handed off" between pods. CephFS can be mounted by multiple writers simultaneously.

> **Important:** You must have your own Ceph server running with the share exported before you can use it.

See the CephFS example for more details.

gitRepo

A `gitRepo` volume is an example of what can be done as a volume plugin. It mounts an empty directory and clones a git repository into it for your pod to use. In the future, such volumes may be moved to an even more decoupled model, rather than extending the Kubernetes API for every such use case.

Here is an example for gitRepo volume:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /mypath
      name: git-volume
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

secret

A `secret` volume is used to pass sensitive information, such as passwords, to pods. You can store secrets in the Kubernetes API and mount them as files for use by pods without coupling to Kubernetes directly. `secret` volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

> **Important:** You must create a secret in the Kubernetes API before you can use it.

Secrets are described in more detail here.

persistentVolumeClaim

A `persistentVolumeClaim` volume is used to mount a PersistentVolume into a pod. PersistentVolumes are a way for users to "claim" durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

See the PersistentVolumes example for more details.

downwardAPI

A `downwardAPI` volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

See the `downwardAPI` volume example for more details.

projected

A `projected` volume maps several existing volume sources into the same directory.

Currently, the following types of volume sources can be projected:

- `secret`
- `downwardAPI`
- `configMap`

All sources are required to be in the same namespace as the pod. For more details, see the all-in-one volume design document.

Example pod with a secret, a downward API, and a configmap.

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - downwardAPI:
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
```

Example pod with multiple secrets with a non-default permission mode set.

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```

Each projected volume source is listed in the spec under `sources`. The parameters are nearly the same with two exceptions:

- For secrets, the `secretName` field has been changed to `name` to be consistent with ConfigMap naming.
- The `defaultMode` can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the `mode` for each individual projection.

AzureFileVolume

A `AzureFileVolume` is used to mount a Microsoft Azure File Volume (SMB 2.1 and 3.0) into a Pod.

More details can be found here.

AzureDiskVolume

A `AzureDiskVolume` is used to mount a Microsoft Azure Data Disk into a Pod.

More details can be found here.

vsphereVolume

> **Prerequisite:** Kubernetes with vSphere Cloud Provider configured. For cloudprovider configuration please refer vSphere getting started guide.

A `vsphereVolume` is used to mount a vSphere VMDK Volume into your Pod. The contents of a volume are preserved when it is unmounted. It supports both VMFS and VSAN datastore.

> **Important:** You must create VMDK using one of the following method before using with POD.

Creating a VMDK volume

Choose one of the following methods to create a VMDK.

Create using vmkfstools | Create using vmware-vdiskmanager

First ssh into ESX, then use the following command to create a VMDK:

```
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk
```

vSphere VMDK Example configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: test-vmdk
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-vmdk
      name: test-volume
  volumes:
  - name: test-volume
    # This VMDK volume must already exist.
    vsphereVolume:
      volumePath: "[DatastoreName] volumes/myDisk"
      fsType: ext4
```

More examples can be found here.

Quobyte

A `Quobyte` volume allows an existing Quobyte volume to be mounted into your pod.

> **Important:** You must have your own Quobyte setup running with the volumes created before you can use it.

See the Quobyte example for more details.

PortworxVolume

A `PortworxVolume` is an elastic block storage layer that runs hyperconverged with Kubernetes. Portworx fingerprints storage in a server, tiers based on capabilities, and aggregates capacity across multiple servers. Portworx runs in-guest in virtual machines or on bare metal Linux nodes.

A `PortworxVolume` can be dynamically created through Kubernetes or it can also be pre-provisioned and referenced inside a Kubernetes pod. Here is an example pod referencing a pre-provisioned PortworxVolume:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /mnt
      name: pxvol
  volumes:
  - name: pxvol
    # This Portworx volume must already exist.
    portworxVolume:
      volumeID: "pxvol"
      fsType: "<fs-type>"
```

> **Important:** Make sure you have an existing PortworxVolume with name `pxvol` before using it in the pod.

More details and examples can be found here.

ScaleIO

ScaleIO is a software-based storage platform that can use existing hardware to create clusters of scalable shared block networked storage. The ScaleIO volume plugin allows deployed pods to access existing ScaleIO volumes (or it can dynamically provision new volumes for persistent volume claims, see ScaleIO Persistent Volumes).

> **Important:** You must have an existing ScaleIO cluster already setup and running with the volumes created before you can use them.

The following is an example pod configuration with ScaleIO:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-0
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: pod-0
    volumeMounts:
    - mountPath: /test-pd
      name: vol-0
  volumes:
  - name: vol-0
    scaleIO:
      gateway: https://localhost:443/api
      system: scaleio
      protectionDomain: sd0
      storagePool: sp1
      volumeName: vol-0
      secretRef:
        name: sio-secret
      fsType: xfs
```

For further detail, please the see the ScaleIO examples.

StorageOS

A `storageos` volume allows an existing StorageOS volume to be mounted into your pod.

StorageOS runs as a container within your Kubernetes environment, making local or attached storage accessible from any node within the Kubernetes cluster. Data can be replicated to protect against node failure. Thin provisioning and compression can improve utilization and reduce cost.

At its core, StorageOS provides block storage to containers, accessible via a file system.

The StorageOS container requires 64-bit Linux and has no additional dependencies. A free developer licence is available.

> **Important:** You must run the StorageOS container on each node that wants to access StorageOS volumes or that will contribute storage capacity to the pool. For installation instructions, consult the StorageOS documentation.

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: redis
    role: master
  name: test-storageos-redis
spec:
  containers:
    - name: master
      image: kubernetes/redis:v1
      env:
        - name: MASTER
          value: "true"
      ports:
        - containerPort: 6379
      volumeMounts:
        - mountPath: /redis-master-data
          name: redis-data
  volumes:
    - name: redis-data
      storageos:
        # The `redis-vol01` volume must already exist within StorageOS in the `def
        volumeName: redis-vol01
        fsType: ext4
```

For more information including Dynamic Provisioning and Persistent Volume Claims, please see the StorageOS examples.

local

This volume type is alpha in 1.7.

A `local` volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be used as a statically created PersistentVolume.

Compared to HostPath volumes, local volumes can be used in a durable manner without manually scheduling pods to nodes, as the system is aware of the volume's node constraints.

However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications.

The following is an example PersistentVolume spec using a `local` volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
  annotations:
        "volume.alpha.kubernetes.io/node-affinity": '{
            "requiredDuringSchedulingIgnoredDuringExecution": {
                "nodeSelectorTerms": [
                    { "matchExpressions": [
                        { "key": "kubernetes.io/hostname",
                          "operator": "In",
                          "values": ["example-node"]
                        }
                    ]}
                 ]}
              }'
spec:
    capacity:
      storage: 100Gi
    accessModes:
    - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: local-storage
    local:
      path: /mnt/disks/ssd1
```

> **Note:** The local PersistentVolume cleanup and deletion requires manual intervention without the external provisioner.

For details on the `local` volume type, see the Local Persistent Storage user guide.

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single pod. The `volumeMounts.subPath` property can be used to specify a sub-path inside the referenced volume instead of its root.

Here is an example of a pod with a LAMP stack (Linux Apache Mysql PHP) using a single, shared volume. The HTML contents are mapped to its `html` folder, and the databases will be stored in its `mysql` folder:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

Resources

The storage media (Disk, SSD, etc.) of an `emptyDir` volume is determined by the medium of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an `emptyDir` or `hostPath` volume can consume, and no isolation between containers or between pods.

In the future, we expect that `emptyDir` and `hostPath` volumes will be able to request a certain amount of space using a resource specification, and to select the type of media to use, for clusters that have several media types.

Out-of-Tree Volume Plugins

In addition to the previously listed volume types, storage vendors may create custom plugins without adding it to the Kubernetes repository. This can be achieved by using the `FlexVolume` plugin.

`FlexVolume` enables users to mount vendor volumes into a pod. The vendor plugin is implemented using a driver, an executable supporting a list of volume commands defined by the `FlexVolume` API. Drivers must be installed in a pre-defined volume plugin path on each node. This is an alpha feature and may change in future.

More details can be found here.

Mount propagation

> **Note:** Mount propagation is an alpha feature in Kubernetes 1.8 and may be redesigned or even removed in future releases.

Mount propagation allows for sharing volumes mounted by a Container to other Containers in the same Pod, or even to other Pods on the same node.

If the MountPropagation feature is disabled, volume mounts in pods are not propagated. That is, Containers run with `private` mount propagation as described in the Linux kernel documentation.

To enable this feature, specify `MountPropagation=true` in the `--feature-gates` command line option. When enabled, the `volumeMounts` field of a Container has a new `mountPropagation` subfield. Its values are:

- `HostToContainer` - This volume mount will receive all subsequent mounts that are mounted to this volume or any of its subdirectories. This is the default mode when the MountPropagation feature is enabled.

  In other words, if the host mounts anything inside the volume mount, the Container will see it mounted there.

  Similarly, if any pod with `Bidirectional` mount propagation to the same volume mounts anything there, the Container with `HostToContainer` mount propagation will see it.

  This mode is equal to `rslave` mount propagation as described in the Linux kernel documentation

- `Bidirectional` - This volume mount behaves the same the `HostToContainer` mount. In addition, all volume mounts created by the Container will be propagated back to the host and to all Containers of all Pods that use the same volume.

  A typical use case for this mode is a Pod with a Flex volume driver or a Pod that needs to mount something on the host using a HostPath volume.

  This mode is equal to `rshared` mount propagation as described in the Linux kernel documentation

Caution: `Bidirectional` mount propagation can be dangerous. It can damage the host operating system and therefore it is allowed only in privileged Containers. Familiarity with Linux kernel behavior is strongly recommended. In addition, any volume mounts created by Containers in Pods must be destroyed (unmounted) by the Containers on termination.

What's next

- Follow an example of deploying WordPress and MySQL with Persistent Volumes.

Persistent Volumes

This document describes the current state of `PersistentVolumes` in Kubernetes. Familiarity with volumes is suggested.

Introduction

Managing storage is a distinct problem from managing compute. The `PersistentVolume` subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this we introduce two new API resources: `PersistentVolume` and `PersistentVolumeClaim`.

A `PersistentVolume` (PV) is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A `PersistentVolumeClaim` (PVC) is a request for storage by a user. It is similar to a pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).

While `PersistentVolumeClaims` allow a user to consume abstract storage resources, it is common that users need `PersistentVolumes` with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of `PersistentVolumes` that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs there is the `StorageClass` resource.

A `StorageClass` provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

Please see the detailed walkthrough with working examples.

Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

Dynamic

When none of the static PVs the administrator created matches a user's `PersistentVolumeClaim`, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on `StorageClasses`: the PVC must request a class and the administrator must have created and configured that class in order for dynamic provisioning to occur. Claims that request the class `""` effectively disable dynamic provisioning for themselves.

Binding

A user creates, or has already created in the case of dynamic provisioning, a `PersistentVolumeClaim` with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, `PersistentVolumeClaim` binds are exclusive, regardless of the mode used to bind them.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a pod. For volumes which support multiple access modes, the user specifies which mode desired when using their claim as a volume in a pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a persistentVolumeClaim in their Pod's volumes block. See below for syntax details.

Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API which allows reclamation of the resource. The reclaim policy for a `PersistentVolume` tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled or Deleted.

Retaining

The Retain reclaim policy allows for manual reclamation of the resource. When the `PersistentVolumeClaim` is deleted, the `PersistentVolume` still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the `PersistentVolume`. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset, or if you want to reuse the same storage asset, create a new `PersistentVolume` with the storage asset definition.

Recycling

If supported by appropriate volume plugin, recycling performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

However, an administrator can configure a custom recycler pod template using the Kubernetes controller manager command line arguments as described here. The custom recycler pod template must contain a `volumes` specification, as shown in the example below:

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-recycler
  namespace: default
spec:
  restartPolicy: Never
  volumes:
  - name: vol
    hostPath:
      path: /any/path/it/will/be/replaced
  containers:
  - name: pv-recycler
    image: "gcr.io/google_containers/busybox"
    command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?* /scrub/.[!.]*
    volumeMounts:
    - name: vol
      mountPath: /scrub
```

However, the particular path specified in the custom recycler pod template in the `volumes` part is replaced with the particular path of the volume that is being recycled.

Deleting

For volume plugins that support the Delete reclaim policy, deletion removes both the `PersistentVolume` object from Kubernetes, as well as deleting the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. Volumes that were dynamically provisioned inherit the reclaim policy of their `StorageClass`, which defaults to Delete. The administrator should configure the `StorageClass` according to users' expectations, otherwise the PV must be edited or patched after it is created. See Change the Reclaim Policy of a PersistentVolume.

Expanding Persistent Volumes Claims

With Kubernetes 1.8, we have added Alpha support for expanding persistent volumes. The current Alpha support was designed to only support volume types that don't need file system resizing (Currently only glusterfs).

Administrator can allow expanding persistent volume claims by setting `ExpandPersistentVolumes` feature gate to true. Administrator should also enable `PersistentVolumeClaimResize` admission plugin to perform additional validations of volumes that can be resized.

Once `PersistentVolumeClaimResize` admission plug-in has been turned on, resizing will only be allowed for storage classes whose `allowVolumeExpansion` field is set to true.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gluster-vol-default
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://192.168.10.100:8080"
  restuser: ""
  secretNamespace: ""
  secretName: ""
allowVolumeExpansion: true
```

Once both feature gate and aforementioned admission plug-in are turned on, an user can request larger volume for their `PersistentVolumeClaim` by simply editing the claim and requesting bigger size. This in turn will trigger expansion of volume that is backing underlying `PersistentVolume`.

Under no circustances a new `PersistentVolume` gets created to satisfy the claim. Kubernetes will attempt to resize existing volume to satisfy the claim.

Types of Persistent Volumes

`PersistentVolume` types are implemented as plugins. Kubernetes currently supports the following plugins:

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- FC (Fibre Channel)
- FlexVolume
- Flocker
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume
- Quobyte Volumes
- HostPath (Single node testing only – local storage is not supported in any way and WILL NOT WORK in a multi-node cluster)
- VMware Photon
- Portworx Volumes

- ScaleIO Volumes
- StorageOS

Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's `capacity` attribute. See the Kubernetes Resource Model to understand the units expected by `capacity` .

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

Access Modes

A `PersistentVolume` can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

The access modes are:

- ReadWriteOnce – the volume can be mounted as read-write by a single node
- ReadOnlyMany – the volume can be mounted read-only by many nodes
- ReadWriteMany – the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany

Important! A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCEPersistentDisk can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes, but not at the same time.

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
FC	✓	✓	-
FlexVolume	✓	✓	-
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
PhotonPersistentDisk	✓	-	-

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	-
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

Class

A PV can have a class, which is specified by setting the `storageClassName` attribute to the name of a `StorageClass`. A PV of a particular class can only be bound to PVCs requesting that class. A PV with no `storageClassName` has no class and can only be bound to PVCs that request no particular class.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of the `storageClassName` attribute. This annotation is still working, however it will become fully deprecated in a future Kubernetes release.

Reclaim Policy

Current reclaim policies are:

- Retain – manual reclamation
- Recycle – basic scrub (`rm -rf /thevolume/*`)
- Delete – associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

Mount Options

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

> **Note:** Not all Persistent volume types support mount options.

The following volume types support mount options:

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- NFS
- iSCSI
- RBD (Ceph Block Device)
- CephFS
- Cinder (OpenStack block storage)
- Glusterfs
- VsphereVolume
- Quobyte Volumes
- VMware Photon

Mount options are not validated, so mount will simply fail if one is invalid.

In the past, the annotation `volume.beta.kubernetes.io/mount-options` was used instead of the `mountOptions` attribute. This annotation is still working, however it will become fully deprecated in a future Kubernetes release.

Phase

A volume will be in one of the following phases:

- Available – a free resource that is not yet bound to a claim
- Bound – the volume is bound to a claim
- Released – the claim has been deleted, but the resource is not yet reclaimed by the cluster
- Failed – the volume has failed its automatic reclamation

The CLI will show the name of the PVC bound to the PV.

PersistentVolumeClaims

Each PVC contains a spec and status, which is the specification and status of the claim.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

Resources

Claims, like pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to both volumes and claims.

Selector

Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- matchLabels - the volume must have a label with this value
- matchExpressions - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include In, NotIn, Exists, and DoesNotExist.

All of the requirements, from both `matchLabels` and `matchExpressions` are ANDed together – they must all be satisfied in order to match.

Class

A claim can request a particular class by specifying the name of a `StorageClass` using the attribute `storageClassName`. Only PVs of the requested class, ones with the same `storageClassName` as the PVC, can be bound to the PVC.

PVCs don't necessarily have to request a class. A PVC with its `storageClassName` set equal to `""` is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to `""`). A PVC with no `storageClassName` is not quite the same and is treated differently by the cluster depending on whether the `DefaultStorageClass` admission plugin is turned on.

- If the admission plugin is turned on, the administrator may specify a default `StorageClass`. All PVCs that have no `storageClassName` can be bound only to PVs of that default. Specifying a default `StorageClass` is done by setting the annotation `storageclass.kubernetes.io/is-default-class` equal to "true" in a `StorageClass` object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default is specified, the admission plugin forbids the creation of all PVCs.
- If the admission plugin is turned off, there is no notion of a default `StorageClass`. All PVCs that have no `storageClassName` can be bound only to PVs that have no class. In this case, the PVCs that have no `storageClassName` are treated the same way as PVCs that have their `storageClassName` set to `""`.

Depending on installation method, a default StorageClass may be deployed to Kubernetes cluster by addon manager during installation.

When a PVC specifies a `selector` in addition to requesting a `StorageClass`, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

> **Note:** Currently, a PVC with a non-empty `selector` can't have a PV dynamically provisioned for it.

In the past, the annotation `volume.beta.kubernetes.io/storage-class` was used instead of `storageClassName` attribute. This annotation is still working, however it won't be supported in a future Kubernetes release.

Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the pod using the claim. The cluster finds the claim in the pod's namespace and uses it to get the `PersistentVolume` backing the claim. The volume is then mounted to the host and into the pod.

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

A Note on Namespaces

`PersistentVolumes` binds are exclusive, and since `PersistentVolumeClaims` are namespaced objects, mounting claims with "Many" modes (`ROX` , `RWX`) is only possible within one namespace.

StorageClasses

Each `StorageClass` contains the fields `provisioner` , `parameters` , and `reclaimPolicy` , which are used when a `PersistentVolume` belonging to the class needs to be dynamically provisioned.

The name of a `StorageClass` object is significant, and is how users can request a particular class. Administrators set the name and other parameters of a class when first creating `StorageClass` objects, and the objects cannot be updated once they are created.

Administrators can specify a default `StorageClass` just for PVCs that don't request any particular class to bind to: see the [`PersistentVolumeClaim` section](#) for details.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
mountOptions:
  - debug
```

Provisioner

Storage classes have a provisioner that determines what volume plugin is used for provisioning PVs. This field must be specified.

Volume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-

Volume Plugin	Internal Provisioner	Config Example
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE
Glusterfs	✓	Glusterfs
iSCSI	-	-
PhotonPersistentDisk	✓	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO

You are not restricted to specifying the "internal" provisioners listed here (whose names are prefixed with "kubernetes.io" and shipped alongside Kubernetes). You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes. Authors of external provisioners have full discretion over where their code lives, how the provisioner is shipped, how it needs to be run, what volume plugin it uses (including Flex), etc. The repository kubernetes-incubator/external-storage houses a library for writing external provisioners that implements the bulk of the specification plus various community-maintained external provisioners.

For example, NFS doesn't provide an internal provisioner, but an external provisioner can be used. Some external provisioners are listed under the repository kubernetes-incubator/external-storage. There are also cases when 3rd party storage vendors provide their own external provisioner.

Reclaim Policy

Persistent Volumes that are dynamically created by a storage class will have the reclaim policy specified in the `reclaimPolicy` field of the class, which can be either `Delete` or `Retain`. If no `reclaimPolicy` is specified when a `StorageClass` object is created, it will default to `Delete`.

Persistent Volumes that are created manually and managed via a storage class will have whatever reclaim policy they were assigned at creation.

Mount Options

Persistent Volumes that are dynamically created by a storage class will have the mount options specified in the `mountOptions` field of the class.

If the volume plugin does not support mount options but mount options are specified, provisioning will fail. Mount options are not validated on neither the class nor PV, so mount of the PV will simply fail if one is invalid.

Parameters

Storage classes have parameters that describe volumes belonging to the storage class. Different parameters may be accepted depending on the `provisioner`. For example, the value `io1`, for the parameter `type`, and the parameter `iopsPerGB` are specific to EBS. When a parameter is omitted, some default is used.

AWS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: us-east-1d, us-east-1c
  iopsPerGB: "10"
```

- `type`: `io1`, `gp2`, `sc1`, `st1`. See AWS docs for details. Default: `gp2`.
- `zone`: AWS zone. If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.

- `zones` : A comma separated list of AWS zone(s). If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `iopsPerGB` : only for `io1` volumes. I/O operations per second per GiB. AWS volume plugin multiplies this with size of requested volume to compute IOPS of the volume and caps it at 20 000 IOPS (maximum supported by AWS, see AWS docs. A string is expected here, i.e. `"10"` , not `10` .
- `encrypted` : denotes whether the EBS volume should be encrypted or not. Valid values are `"true"` or `"false"` . A string is expected here, i.e. `"true"` , not `true` .
- `kmsKeyId` : optional. The full Amazon Resource Name of the key to use when encrypting the volume. If none is supplied but `encrypted` is true, a key is generated by AWS. See AWS docs for valid ARN value.

GCE

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zones: us-central1-a, us-central1-b
```

- `type` : `pd-standard` or `pd-ssd` . Default: `pd-standard`
- `zone` : GCE zone. If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.
- `zones` : A comma separated list of GCE zone(s). If neither `zone` nor `zones` is specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node. `zone` and `zones` parameters must not be used at the same time.

Glusterfs

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

- `resturl` : Gluster REST service/Heketi service url which provision gluster volumes on demand. The general format should be `IPaddress:Port` and this is a mandatory parameter for GlusterFS dynamic provisioner. If Heketi service is exposed as a routable service in openshift/kubernetes setup, this can have a format similar to `http://heketi-storage-project.cloudapps.mystorage.com` where the fqdn is a resolvable heketi service url.
- `restauthenabled` : Gluster REST service authentication boolean that enables authentication to the REST server. If this value is 'true', `restuser` and `restuserkey` or `secretNamespace` + `secretName` have to be filled. This option is deprecated, authentication is enabled when any of `restuser`, `restuserkey`, `secretName` or `secretNamespace` is specified.
- `restuser` : Gluster REST service/Heketi user who has access to create volumes in the Gluster Trusted Pool.
- `restuserkey` : Gluster REST service/Heketi user's password which will be used for authentication to the REST server. This parameter is deprecated in favor of `secretNamespace` + `secretName`.
- `secretNamespace`, `secretName` : Identification of Secret instance that contains user password to use when talking to Gluster REST service. These parameters are optional, empty password

will be used when both `secretNamespace` and `secretName` are omitted. The provided secret must have type "kubernetes.io/glusterfs", e.g. created in this way:

```
$ kubectl create secret generic heketi-secret --type="kubernetes.io/glusterfs"
--from-literal=key='opensesame' --namespace=default
```

Example of a secret can be found in glusterfs-provisioning-secret.yaml.

- `clusterid`: `630372ccdc720a92c681fb928f27b53f` is the ID of the cluster which will be used by Heketi when provisioning the volume. It can also be a list of clusterids, for ex: "8452344e2becec931ece4e33c4674e4e,42982310de6c63381718ccfa6d8cf397". This is an optional parameter.
- `gidMin`, `gidMax` : The minimum and maximum value of GID range for the storage class. A unique value (GID) in this range (gidMin-gidMax) will be used for dynamically provisioned volumes. These are optional values. If not specified, the volume will be provisioned with a value between 2000-2147483647 which are defaults for gidMin and gidMax respectively.
- `volumetype` : The volume type and its parameters can be configured with this optional value. If the volume type is not mentioned, it's up to the provisioner to decide the volume type. For example: 'Replica volume': `volumetype: replicate:3` where '3' is replica count. 'Disperse/EC volume': `volumetype: disperse:4:2` where '4' is data and '2' is the redundancy count. 'Distribute volume': `volumetype: none`

  For available volume types and administration options, refer to the Administration Guide.

  For further reference information, see How to configure Heketi.

  When persistent volumes are dynamically provisioned, the Gluster plugin automatically creates an endpoint and a headless service in the name `gluster-dynamic-<claimname>`. The dynamic endpoint and service are automatically deleted when the persistent volume claim is deleted.

OpenStack Cinder

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  type: fast
  availability: nova
```

- `type` : VolumeType created in Cinder. Default is empty.
- `availability` : Availability Zone. If not specified, volumes are generally round-robin-ed across all active zones where Kubernetes cluster has a node.

vSphere

1. Create a persistent volume with a user specified disk format.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: zeroedthick
```

- `diskformat` : `thin` , `zeroedthick` and `eagerzeroedthick` . Default: `"thin"` .

1. Create a persistent volume with a disk format on a user specified datastore.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
    diskformat: zeroedthick
    datastore: VSANDatastore
```

- `diskformat` : `thin` , `zeroedthick` and `eagerzeroedthick` . Default: `"thin"` .

- `datastore` : The user can also specify the datastore in the Storageclass. The volume will be created on the datastore specified in the storage class which in this case is `VSANDatastore` . This field is optional. If not specified as in previous YAML description, the volume will be created on the datastore specified in the vsphere config file used to initialize the vSphere Cloud Provider.

1. Create a persistent volume with user specified VSAN storage capabilities.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: vsan-policy-fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: thin
  hostFailuresToTolerate: "1"
  diskStripes: "2"
  cacheReservation: "20"
  datastore: VSANDatastore
```

- Here, the user can specify VSAN storage capabilities for dynamic volume provisioning inside Kubernetes.

- Storage Policies capture storage requirements, such as performance and availability, for persistent volumes. These policies determine how the container volume storage objects are provisioned and allocated within the datastore to guarantee the requested Quality of Service. Storage policies are composed of storage capabilities, typically represented by a key-value pair. The key is a specific property that the datastore can offer and the value is a metric, or a range, that the datastore guarantees for a provisioned object, such as a container volume backed by a virtual disk.

- As described in official documentation, VSAN exposes multiple storage capabilities. The below table lists VSAN storage capabilities that are currently supported by vSphere Cloud Provider.

Storage Capability Name	Description
cacheReservation	Flash read cache reservation
diskStripes	Number of disk stripes per object

Storage Capability Name	Description
forceProvisioning	Force provisioning
hostFailuresToTolerate	Number of failures to tolerate
iopsLimit	IOPS limit for object
objectSpaceReservation	Object space reservation

vSphere Infrastructure(VI) administrator can specify storage requirements for applications in terms of storage capabilities while creating a storage class inside Kubernetes. Please note that while creating a StorageClass, administrator should specify storage capability names used in the table above as these names might differ from the ones used by VSAN. For example - Number of disk stripes per object is referred to as stripeWidth in VSAN documentation however vSphere Cloud Provider uses a friendly name diskStripes.

You can see vSphere example for more details.

Ceph RBD

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

- `monitors` : Ceph monitors, comma delimited. This parameter is required.
- `adminId` : Ceph client ID that is capable of creating images in the pool. Default is “admin”.

- `adminSecretNamespace` : The namespace for `adminSecret` . Default is “default”.
- `adminSecret` : Secret Name for `adminId` . This parameter is required. The provided secret must have type “kubernetes.io/rbd”.
- `pool` : Ceph RBD pool. Default is “rbd”.
- `userId` : Ceph client ID that is used to map the RBD image. Default is the same as `adminId` .
- `userSecretName` : The name of Ceph Secret for `userId` to map RBD image. It must exist in the same namespace as PVCs. This parameter is required. The provided secret must have type “kubernetes.io/rbd”, e.g. created in this way:
  ```
  $ kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" --from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ==' --namespace=kube-system
  ```
- `fsType` : fsType that is supported by kubernetes. Default: `"ext4"` .
- `imageFormat` : Ceph RBD image format, “1” or “2”. Default is “1”.
- `imageFeatures` : This parameter is optional and should only be used if you set `imageFormat` to “2”. Currently supported features are `layering` only. Default is “”, and no features are turned on.

Quobyte

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: slow
provisioner: kubernetes.io/quobyte
parameters:
    quobyteAPIServer: "http://138.68.74.142:7860"
    registry: "138.68.74.142:7861"
    adminSecretName: "quobyte-admin-secret"
    adminSecretNamespace: "kube-system"
    user: "root"
    group: "root"
    quobyteConfig: "BASE"
    quobyteTenant: "DEFAULT"
```

- `quobyteAPIServer` : API Server of Quobyte in the format `http(s)://api-server:7860`

- `registry` : Quobyte registry to use to mount the volume. You can specify the registry as `<host>:<port>` pair or if you want to specify multiple registries you just have to put a comma between them e.q. `<host1>:<port>,<host2>:<port>,<host3>:<port>` . The host can be an IP address or if you have a working DNS you can also provide the DNS names.
- `adminSecretNamespace` : The namespace for `adminSecretName` . Default is "default".
- `adminSecretName` : secret that holds information about the Quobyte user and the password to authenticate against the API server. The provided secret must have type "kubernetes.io/quobyte", e.g. created in this way:
  ```
  $ kubectl create secret generic quobyte-admin-secret --type="kubernetes.io/quobyte" --from-literal=key='opensesame' --namespace=kube-system
  ```
- `user` : maps all access to this user. Default is "root".
- `group` : maps all access to this group. Default is "nfsnobody".
- `quobyteConfig` : use the specified configuration to create the volume. You can create a new configuration or modify an existing one with the Web console or the quobyte CLI. Default is "BASE".
- `quobyteTenant` : use the specified tenant ID to create/delete the volume. This Quobyte tenant has to be already present in Quobyte. Default is "DEFAULT".

Azure Disk

Azure Unmanaged Disk Storage Class

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

- `skuName` : Azure storage account Sku tier. Default is empty.
- `location` : Azure storage account location. Default is empty.
- `storageAccount` : Azure storage account name. If a storage account is provided, it must reside in the same resource group as the cluster, and `location` is ignored. If a storage account is not provided, a new storage account will be created in the same resource group as the cluster.

New Azure Disk Storage Class (starting from v1.7.2)

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/azure-disk
parameters:
  storageaccounttype: Standard_LRS
  kind: Shared
```

- `storageaccounttype` : Azure storage account Sku tier. Default is empty.
- `kind` : Possible values are `shared` (default), `dedicated`, and `managed`. When `kind` is `shared`, all unmanaged disks are created in a few shared storage accounts in the same resource group as the cluster. When `kind` is `dedicated`, a new dedicated storage account will be created for the new unmanaged disk in the same resource group as the cluster.
- Premium VM can attach both Standard_LRS and Premium_LRS disks, while Standard VM can only attach Standard_LRS disks.
- Managed VM can only attach managed disks and unmanaged VM can only attach unmanaged disks.

Azure File

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
parameters:
  skuName: Standard_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

- `skuName` : Azure storage account Sku tier. Default is empty.
- `location` : Azure storage account location. Default is empty.
- `storageAccount` : Azure storage account name. Default is empty. If a storage account is not provided, all storage accounts associated with the resource group are searched to find one that matches `skuName` and `location` . If a storage account is provided, it must reside in the same resource group as the cluster, and `skuName` and `location` are ignored.

During provision, a secret is created for mounting credentials. If the cluster has enabled both RBAC and Controller Roles, add the `create` permission of resource `secret` for clusterrole `system:controller:persistent-volume-binder` .

Portworx Volume

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: portworx-io-priority-high
provisioner: kubernetes.io/portworx-volume
parameters:
  repl: "1"
  snap_interval:   "70"
  io_priority:  "high"
```

- `fs` : filesystem to be laid out: [none/xfs/ext4] (default: `ext4`).
- `block_size` : block size in Kbytes (default: `32`).

- `repl` : number of synchronous replicas to be provided in the form of replication factor [1..3] (default: `1`) A string is expected here i.e. `"1"` and not `1` .
- `io_priority` : determines whether the volume will be created from higher performance or a lower priority storage [high/medium/low] (default: `low`).
- `snap_interval` : clock/time interval in minutes for when to trigger snapshots. Snapshots are incremental based on difference with the prior snapshot, 0 disables snaps (default: `0`). A string is expected here i.e. `"70"` and not `70` .
- `aggregation_level` : specifies the number of chunks the volume would be distributed into, 0 indicates a non-aggregated volume (default: `0`). A string is expected here i.e. `"0"` and not `0`
- `ephemeral` : specifies whether the volume should be cleaned-up after unmount or should be persistent. `emptyDir` use case can set this value to true and `persistent volumes` use case such as for databases like Cassandra should set to false, [true/false] (default `false`). A string is expected here i.e. `"true"` and not `true` .

ScaleIO

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/scaleio
parameters:
  gateway: https://192.168.99.200:443/api
  system: scaleio
  protectionDomain: pd0
  storagePool: sp1
  storageMode: ThinProvisionned
  secretRef: sio-secret
  readOnly: false
  fsType: xfs
```

- `provisioner` : attribute is set to `kubernetes.io/scaleio`
- `gateway` : address to a ScaleIO API gateway (required)

- `system` : the name of the ScaleIO system (required)
- `protectionDomain` : the name of the ScaleIO protection domain (required)
- `storagePool` : the name of the volume storage pool (required)
- `storageMode` : the storage provision mode: `ThinProvisionned` (default) or `ThickProvisionned`
- `secretRef` : reference to a configured Secret object (required)
- `readOnly` : specifies the access mode to the mounted volume (default false)
- `fsType` : the file system to use for the volume (default ext4)

The ScaleIO Kubernetes volume plugin requires a configured Secret object. The secret must be created with type `kubernetes.io/scaleio` and use the same namespace value as that of the PVC where it is referenced as shown in the following command:

```
$> kubectl create secret generic sio-secret --type="kubernetes.io/scaleio" --from-
```

StorageOS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/storageos
parameters:
  pool: default
  description: Kubernetes volume
  fsType: ext4
  adminSecretNamespace: default
  adminSecretName: storageos-secret
```

- `pool` : The name of the StorageOS distributed capacity pool to provision the volume from. Uses the `default` pool which is normally present if not specified.

- `description`: The description to assign to volumes that were created dynamically. All volume descriptions will be the same for the storage class, but different storage classes can be used to allow descriptions for different use cases. Defaults to `Kubernetes volume`.
- `fsType`: The default filesystem type to request. Note that user-defined rules within StorageOS may override this value. Defaults to `ext4`.
- `adminSecretNamespace`: The namespace where the API configuration secret is located. Required if adminSecretName set.
- `adminSecretName`: The name of the secret to use for obtaining the StorageOS API credentials. If not specified, default values will be attempted.

The StorageOS Kubernetes volume plugin can use a Secret object to specify an endpoint and credentials to access the StorageOS API. This is only required when the defaults have been changed. The secret must be created with type `kubernetes.io/storageos` as shown in the following command:

```
$ kubectl create secret generic storageos-secret --type="kubernetes.io/storageos"
```

Secrets used for dynamically provisioned volumes may be created in any namespace and referenced with the `adminSecretNamespace` parameter. Secrets used by pre-provisioned volumes must be created in the same namespace as the PVC that references it.

Writing Portable Configuration

If you’re writing configuration templates or examples that run on a wide range of clusters and need persistent storage, we recommend that you use the following pattern:

- Do include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
- Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.

- Give the user the option of providing a storage class name when instantiating the template.
  - If the user provides a storage class name, and the cluster is version 1.4 or newer, put that value into the `volume.beta.kubernetes.io/storage-class` annotation of the PVC. This will cause the PVC to match the right storage class if the cluster has StorageClasses enabled by the admin.
  - If the user does not provide a storage class name or the cluster is version 1.3, then instead put a `volume.alpha.kubernetes.io/storage-class: default` annotation on the PVC.
    - This will cause a PV to be automatically provisioned for the user with sane default characteristics on some clusters.
    - Despite the word `alpha` in the name, the code behind this annotation has `beta` level support.
    - Do not use `volume.beta.kubernetes.io/storage-class:` with any value including the empty string since it will prevent DefaultStorageClass admission controller from running if enabled.
- In your tooling, do watch for PVCs that are not getting bound after some time and surface this to the user, as this may indicate that the cluster has no dynamic storage support (in which case the user should create a matching PV) or the cluster has no storage system (in which case the user cannot deploy config requiring PVCs).
- In the future, we expect most clusters to have `DefaultStorageClass` enabled, and to have some form of storage available. However, there may not be any storage class names which work on all clusters, so continue to not set one by default. At some point, the alpha annotation will cease to have meaning, but the unset `storageClass` field on the PVC will have the desired effect.

Cluster Administration Overview

The cluster administration overview is for anyone creating or administering a Kubernetes cluster. It assumes some familiarity with concepts in the User Guide.

Planning a cluster

See the guides in Picking the Right Solution for examples of how to plan, set up, and configure Kubernetes clusters. The solutions listed in this article are called *distros*.

Before choosing a guide, here are some considerations:

- Do you just want to try out Kubernetes on your computer, or do you want to build a high-availability, multi-node cluster? Choose distros best suited for your needs.
- **If you are designing for high-availability**, learn about configuring clusters in multiple zones.
- Will you be using **a hosted Kubernetes cluster**, such as Google Container Engine (GKE), or **hosting your own cluster**?
- Will your cluster be **on-premises**, or **in the cloud (IaaS)**? Kubernetes does not directly support hybrid clusters. Instead, you can set up multiple clusters.
- **If you are configuring Kubernetes on-premises**, consider which networking model fits best. One option for custom networking is *OpenVSwitch GRE/VxLAN networking*, which uses OpenVSwitch to set up networking between pods across Kubernetes nodes.
- Will you be running Kubernetes on **“bare metal” hardware** or on **virtual machines (VMs)**?
- Do you **just want to run a cluster**, or do you expect to do **active development of Kubernetes project code**? If the latter, choose a actively-developed distro. Some distros only use binary

releases, but offer a greater variety of choices.

- Familiarize yourself with the components needed to run a cluster.

Note: Not all distros are actively maintained. Choose distros which have been tested with a recent version of Kubernetes.

If you are using a guide involving Salt, see Configuring Kubernetes with Salt.

Managing a cluster

- Managing a cluster describes several topics related to the lifecycle of a cluster: creating a new cluster, upgrading your cluster's master and worker nodes, performing node maintenance (e.g. kernel upgrades), and upgrading the Kubernetes API version of a running cluster.
- Learn how to manage nodes.
- Learn how to set up and manage the resource quota for shared clusters.

Securing a cluster

- Kubernetes Container Environment describes the environment for Kubelet managed containers on a Kubernetes node.
- Controlling Access to the Kubernetes API describes how to set up permissions for users and service accounts.
- Authenticating explains authentication in Kubernetes, including the various authentication options.
- Authorization is separate from authentication, and controls how HTTP calls are handled.
- Using Admission Controllers explains plug-ins which intercepts requests to the Kubernetes API server after authentication and authorization.
- Using Sysctls in a Kubernetes Cluster describes to an administrator how to use the `sysctl` command-line tool to set kernel parameters .
- Auditing describes how to interact with Kubernetes' audit logs.

Securing the kubelet

- Master-Node communication
- TLS bootstrapping
- Kubelet authentication/authorization

Optional Cluster Services

- DNS Integration with SkyDNS describes how to resolve a DNS name directly to a Kubernetes service.
- Logging and Monitoring Cluster Activity explains how logging in Kubernetes works and how to implement it.

Cloud Providers

This page explains how to manage Kubernetes running on a specific cloud provider.

- AWS
  - Load Balancers

AWS

This section describes all the possible configurations which can be used when running Kubernetes on Amazon Web Services.

Load Balancers

You can setup external load balancers to use specific features in AWS by configuring the annotations as shown below.

```
apiVersion: v1
kind: Service
metadata:
  name: example
  namespace: kube-system
  labels:
    run: example
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:xx-xxxx-x:
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 5556
    protocol: TCP
  selector:
    app: example
```

Different settings can be applied to a load balancer service in AWS using *annotations*. The following describes the annotations supported on AWS ELBs:

- `service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval` : Used to specify access log emit interval.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-enabled` : Used on the service to enable or disable access logs.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name` : Used to specify access log s3 bucket name.
- `service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix` : Used to specify access log s3 bucket prefix.
- `service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags` : Used on the service to specify a comma-separated list of key-value pairs which will be recorded as additional tags in the ELB. For example: `"Key1=Val1,Key2=Val2,KeyNoVal1=,KeyNoVal2"` .
- `service.beta.kubernetes.io/aws-load-balancer-backend-protocol` : Used on the service to specify the protocol spoken by the backend (pod) behind a listener. If `http` (default) or `https` , an HTTPS listener that terminates the connection and parses headers is created. If set to `ssl` or `tcp` , a "raw" SSL listener is used. If set to `http` and `aws-load-balancer-ssl-cert` is not used then a HTTP listener is used.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-cert` : Used on the service to request a secure listener. Value is a valid certificate ARN. For more, see ELB Listener Config CertARN is an IAM or CM certificate ARN, e.g.
  ```
  arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012
  ```
  .
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled` : Used on the service to enable or disable connection draining.
- `service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout` : Used on the service to specify a connection draining timeout.

- `service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout` : Used on the service to specify the idle connection timeout.
- `service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled` : Used on the service to enable or disable cross-zone load balancing.
- `service.beta.kubernetes.io/aws-load-balancer-extra-security-groups` : Used one the service to specify additional security groups to be added to ELB created
- `service.beta.kubernetes.io/aws-load-balancer-internal` : Used on the service to indicate that we want an internal ELB.
- `service.beta.kubernetes.io/aws-load-balancer-proxy-protocol` : Used on the service to enable the proxy protocol on an ELB. Right now we only accept the value `*` which means enable the proxy protocol on all ELB backends. In the future we could adjust this to allow setting the proxy protocol only on certain backends.
- `service.beta.kubernetes.io/aws-load-balancer-ssl-ports` : Used on the service to specify a comma-separated list of ports that will use SSL/HTTPS listeners. Defaults to `*` (all)

The information for the annotations for AWS is taken from the comments on aws.go

Managing Resources

You've deployed your application and exposed it via a service. Now what? Kubernetes provides a number of tools to help you manage your application deployment, including scaling and updating. Among the features that we will discuss in more depth are configuration files and labels.

You can find all the files for this example in our docs repo here.

Organizing resource configurations

Many applications require multiple resources to be created, such as a Deployment and a Service. Management of multiple resources can be simplified by grouping them together in the same file (separated by `---` in YAML). For example:

nginx-app.yaml

nginx-app.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
  labels:
    app: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Multiple resources can be created the same way as a single resource:

```
$ kubectl create -f docs/user-guide/nginx-app.yaml
service "my-nginx-svc" created
deployment "my-nginx" created
```

The resources will be created in the order they appear in the file. Therefore, it's best to specify the service first, since that will ensure the scheduler can spread the pods associated with the service as they are created by the controller(s), such as Deployment.

`kubectl create` also accepts multiple `-f` arguments:

```
$ kubectl create -f docs/user-guide/nginx/nginx-svc.yaml -f docs/user-guide/nginx/
```

And a directory can be specified rather than or in addition to individual files:

```
$ kubectl create -f docs/user-guide/nginx/
```

`kubectl` will read any files with suffixes `.yaml`, `.yml`, or `.json`.

It is a recommended practice to put resources related to the same microservice or application tier into the same file, and to group all of the files associated with your application in the same directory. If the tiers of your application bind to each other using DNS, then you can then simply deploy all of the components of your stack en masse.

A URL can also be specified as a configuration source, which is handy for deploying directly from configuration files checked into github:

```
$ kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/master
deployment "nginx-deployment" created
```

Bulk operations in kubectl

Resource creation isn't the only operation that `kubectl` can perform in bulk. It can also extract resource names from configuration files in order to perform other operations, in particular to delete the same resources you created:

```
$ kubectl delete -f docs/user-guide/nginx/
deployment "my-nginx" deleted
service "my-nginx-svc" deleted
```

In the case of just two resources, it's also easy to specify both on the command line using the resource/name syntax:

```
$ kubectl delete deployments/my-nginx services/my-nginx-svc
```

For larger numbers of resources, you'll find it easier to specify the selector (label query) specified using `-l` or `--selector`, to filter resources by their labels:

```
$ kubectl delete deployment,services -l app=nginx
deployment "my-nginx" deleted
service "my-nginx-svc" deleted
```

Because `kubectl` outputs resource names in the same syntax it accepts, it's easy to chain operations using `$()` or `xargs`:

```
$ kubectl get $(kubectl create -f docs/user-guide/nginx/ -o name | grep service)
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
my-nginx-svc   10.0.0.208   <pending>     80/TCP       0s
```

With the above commands, we first create resources under docs/user-guide/nginx/ and print the resources created with `-o name` output format (print each resource as resource/name). Then we `grep` only the "service", and then print it with `kubectl get`.

If you happen to organize your resources across several subdirectories within a particular directory, you can recursively perform the operations on the subdirectories also, by specifying `--recursive` or `-R` alongside the `--filename,-f` flag.

For instance, assume there is a directory `project/k8s/development` that holds all of the manifests needed for the development environment, organized by resource type:

```
project/k8s/development
├── configmap
│   └── my-configmap.yaml
├── deployment
│   └── my-deployment.yaml
└── pvc
    └── my-pvc.yaml
```

By default, performing a bulk operation on `project/k8s/development` will stop at the first level of the directory, not processing any subdirectories. If we had tried to create the resources in this directory using the following command, we would have encountered an error:

```
$ kubectl create -f project/k8s/development
error: you must provide one or more resources by argument or filename (.json|.yaml
```

Instead, specify the `--recursive` or `-R` flag with the `--filename,-f` flag as such:

```
$ kubectl create -f project/k8s/development --recursive
configmap "my-config" created
deployment "my-deployment" created
persistentvolumeclaim "my-pvc" created
```

The `--recursive` flag works with any operation that accepts the `--filename,-f` flag such as: `kubectl {create,get,delete,describe,rollout} etc.`

The `--recursive` flag also works when multiple `-f` arguments are provided:

```
$ kubectl create -f project/k8s/namespaces -f project/k8s/development --recursive
namespace "development" created
namespace "staging" created
configmap "my-config" created
deployment "my-deployment" created
persistentvolumeclaim "my-pvc" created
```

If you're interested in learning more about `kubectl`, go ahead and read kubectl Overview.

Using labels effectively

The examples we've used so far apply at most a single label to any resource. There are many scenarios where multiple labels should be used to distinguish sets from one another.

For instance, different applications would use different values for the `app` label, but a multi-tier application, such as the guestbook example, would additionally need to distinguish each tier. The frontend could carry the following labels:

```
     labels:
        app: guestbook
        tier: frontend
```

while the Redis master and slave would have different `tier` labels, and perhaps even an additional `role` label:

```
     labels:
        app: guestbook
        tier: backend
        role: master
```

and

```
     labels:
        app: guestbook
        tier: backend
        role: slave
```

The labels allow us to slice and dice our resources along any dimension specified by a label:

```
$ kubectl create -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
$ kubectl get pods -Lapp -Ltier -Lrole
NAME                           READY     STATUS    RESTARTS   AGE       APP
guestbook-fe-4nlpb             1/1       Running   0          1m        guestbook
guestbook-fe-ght6d             1/1       Running   0          1m        guestbook
guestbook-fe-jpy62             1/1       Running   0          1m        guestbook
guestbook-redis-master-5pg3b   1/1       Running   0          1m        guestbook
guestbook-redis-slave-2q2yf    1/1       Running   0          1m        guestbook
guestbook-redis-slave-qgazl    1/1       Running   0          1m        guestbook
my-nginx-divi2                 1/1       Running   0          29m       nginx
my-nginx-o0ef1                 1/1       Running   0          29m       nginx
$ kubectl get pods -lapp=guestbook,role=slave
NAME                          READY     STATUS    RESTARTS   AGE
guestbook-redis-slave-2q2yf   1/1       Running   0          3m
guestbook-redis-slave-qgazl   1/1       Running   0          3m
```

Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. It is common practice to deploy a *canary* of a new application release (specified via image tag in the pod template) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out.

For instance, you can use a `track` label to differentiate different releases.

The primary, stable release would have a `track` label with value as `stable`:

```
     name: frontend
     replicas: 3
     ...
     labels:
        app: guestbook
        tier: frontend
        track: stable
     ...
     image: gb-frontend:v3
```

and then you can create a new release of the guestbook frontend that carries the `track` label with different value (i.e. `canary`), so that two sets of pods would not overlap:

```
     name: frontend-canary
     replicas: 1
     ...
     labels:
        app: guestbook
        tier: frontend
        track: canary
     ...
     image: gb-frontend:v4
```

The frontend service would span both sets of replicas by selecting the common subset of their labels (i.e. omitting the `track` label), so that the traffic will be redirected to both applications:

```
  selector:
     app: guestbook
     tier: frontend
```

You can tweak the number of replicas of the stable and canary releases to determine the ratio of each release that will receive live production traffic (in this case, 3:1). Once you're confident, you can update the stable track to the new application release and remove the canary one.

For a more concrete example, check the tutorial of deploying Ghost.

Updating labels

Sometimes existing pods and other resources need to be relabeled before creating new resources. This can be done with `kubectl label`. For example, if you want to label all your nginx pods as frontend tier, simply run:

```
$ kubectl label pods -l app=nginx tier=fe
pod "my-nginx-2035384211-j5fhi" labeled
pod "my-nginx-2035384211-u2c7e" labeled
pod "my-nginx-2035384211-u3t6x" labeled
```

This first filters all pods with the label "app=nginx", and then labels them with the "tier=fe". To see the pods you just labeled, run:

```
$ kubectl get pods -l app=nginx -L tier
NAME                        READY     STATUS    RESTARTS   AGE       TIER
my-nginx-2035384211-j5fhi   1/1       Running   0          23m       fe
my-nginx-2035384211-u2c7e   1/1       Running   0          23m       fe
my-nginx-2035384211-u3t6x   1/1       Running   0          23m       fe
```

This outputs all "app=nginx" pods, with an additional label column of pods' tier (specified with `-L` or `--label-columns`).

For more information, please see labels and kubectl label document.

Updating annotations

Sometimes you would want to attach annotations to resources. Annotations are arbitrary non-identifying metadata for retrieval by API clients such as tools, libraries, etc. This can be done with `kubectl annotate`. For example:

```
$ kubectl annotate pods my-nginx-v4-9gw19 description='my frontend running nginx'
$ kubectl get pods my-nginx-v4-9gw19 -o yaml
apiversion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
...
```

For more information, please see annotations and kubectl annotate document.

Scaling your application

When load on your application grows or shrinks, it's easy to scale with `kubectl`. For instance, to decrease the number of nginx replicas from 3 to 1, do:

```
$ kubectl scale deployment/my-nginx --replicas=1
deployment "my-nginx" scaled
```

Now you only have one pod managed by the deployment.

```
$ kubectl get pods -l app=nginx
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-2035384211-j5fhi   1/1       Running   0          30m
```

To have the system automatically choose the number of nginx replicas as needed, ranging from 1 to 3, do:

```
$ kubectl autoscale deployment/my-nginx --min=1 --max=3
deployment "my-nginx" autoscaled
```

Now your nginx replicas will be scaled up and down as needed, automatically.

For more information, please see kubectl scale, kubectl autoscale and horizontal pod autoscaler document.

In-place updates of resources

Sometimes it’s necessary to make narrow, non-disruptive updates to resources you’ve created.

kubectl apply

It is suggested to maintain a set of configuration files in source control (see configuration as code), so that they can be maintained and versioned along with the code for the resources they configure. Then, you can use `kubectl apply` to push your configuration changes to the cluster.

This command will compare the version of the configuration that you’re pushing with the previous version and apply the changes you’ve made, without overwriting any automated changes to properties you haven’t specified.

```
$ kubectl apply -f docs/user-guide/nginx/nginx-deployment.yaml
deployment "my-nginx" configured
```

Note that `kubectl apply` attaches an annotation to the resource in order to determine the changes to the configuration since the previous invocation. When it’s invoked, `kubectl apply` does a three-way diff between the previous configuration, the provided input and the current configuration of the resource, in order to determine how to modify the resource.

Currently, resources are created without this annotation, so the first invocation of `kubectl apply` will fall back to a two-way diff between the provided input and the current configuration of the resource. During this first invocation, it cannot detect the deletion of properties set when the resource was created. For this reason, it will not remove them.

All subsequent calls to `kubectl apply`, and other commands that modify the configuration, such as `kubectl replace` and `kubectl edit`, will update the annotation, allowing subsequent calls to `kubectl apply` to detect and perform deletions using a three-way diff.

Note: To use apply, always create resource initially with either `kubectl apply` or `kubectl create --save-config`.

kubectl edit

Alternatively, you may also update resources with `kubectl edit`:

```
$ kubectl edit deployment/my-nginx
```

This is equivalent to first `get` the resource, edit it in text editor, and then `apply` the resource with the updated version:

```
$ kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
$ vi /tmp/nginx.yaml
# do some edit, and then save the file
$ kubectl apply -f /tmp/nginx.yaml
deployment "my-nginx" configured
$ rm /tmp/nginx.yaml
```

This allows you to do more significant changes more easily. Note that you can specify the editor with your `EDITOR` or `KUBE_EDITOR` environment variables.

For more information, please see kubectl edit document.

kubectl patch

You can use `kubectl patch` to update API objects in place. This command supports JSON patch, JSON merge patch, and strategic merge patch. See Update API Objects in Place Using kubectl patch and kubectl patch.

Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may just want to make a recursive change immediately, such as to fix broken pods created by a Deployment. To change such fields, use `replace --force`, which deletes and re-creates the resource. In this case, you can simply modify your original configuration file:

```
$ kubectl replace -f docs/user-guide/nginx/nginx-deployment.yaml --force
deployment "my-nginx" deleted
deployment "my-nginx" replaced
```

Updating your application without a service outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag, as in the canary deployment scenario above. `kubectl` supports several update operations, each of which is applicable to different scenarios.

We'll guide you through how to create and update applications with Deployments. If your deployed application is managed by Replication Controllers, you should read how to use `kubectl rolling-update` instead.

Let's say you were running version 1.7.9 of nginx:

```
$ kubectl run my-nginx --image=nginx:1.7.9 --replicas=3
deployment "my-nginx" created
```

To update to version 1.9.1, simply change `.spec.template.spec.containers[0].image` from `nginx:1.7.9` to `nginx:1.9.1`, with the kubectl commands we learned above.

```
$ kubectl edit deployment/my-nginx
```

That's it! The Deployment will declaratively update the deployed nginx application progressively behind the scene. It ensures that only a certain number of old replicas may be down while they are being updated, and only a certain number of new replicas may be created above the desired number of pods. To learn more details about it, visit Deployment page.

What's next?

- Learn about how to use `kubectl` for application introspection and debugging.
- Configuration Best Practices and Tips

Cluster Networking

Kubernetes approaches networking somewhat differently than Docker does by default. There are 4 distinct networking problems to solve:

1. Highly-coupled container-to-container communications: this is solved by pods and `localhost` communications.
2. Pod-to-Pod communications: this is the primary focus of this document.
3. Pod-to-Service communications: this is covered by services.
4. External-to-Service communications: this is covered by services.

Summary

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. We give every pod its own IP address so you do not need to explicitly create links between pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

To achieve this we must impose some requirements on how you set up your cluster networking.

Docker model

Before discussing the Kubernetes approach to networking, it is worthwhile to review the "normal" way that networking works with Docker. By default, Docker uses host-private networking. It creates a virtual bridge, called `docker0` by default, and allocates a subnet from one of the private address blocks defined in RFC1918 for that bridge. For each container that Docker creates, it allocates a virtual Ethernet device (called `veth`) which is attached to the bridge. The veth is mapped to appear as `eth0` in the container, using Linux namespaces. The in-container `eth0` interface is given an IP address from the bridge's address range.

The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines can not reach each other - in fact they may end up with the exact same network ranges and IP addresses.

In order for Docker containers to communicate across nodes, they must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or else be allocated ports dynamically.

Kubernetes model

Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control. Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- all containers can communicate with all other containers without NAT
- all nodes can communicate with all containers (and vice-versa) without NAT
- the IP that a container sees itself as is the same IP that others see it as

What this means in practice is that you can not just take two computers running Docker and expect Kubernetes to work. You must ensure that the fundamental requirements are met.

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Until now this document has talked about containers. In reality, Kubernetes applies IP addresses at the `Pod` scope - containers within a `Pod` share their network namespaces - including their IP address. This means that containers within a `Pod` can all reach each other's ports on `localhost`. This does imply that containers within a `Pod` must coordinate port usage, but this is no different than processes in a VM. We call this the "IP-per-pod" model. This is implemented in Docker as a "pod container" which holds the network namespace open while "app containers" (the things the user specified) join that namespace with Docker's `--net=container:<id>` function.

As with Docker, it is possible to request host ports, but this is reduced to a very niche operation. In this case a port will be allocated on the host `Node` and traffic will be forwarded to the `Pod`. The `Pod` itself is blind to the existence or non-existence of host ports.

How to achieve this

There are a number of ways that this network model can be implemented. This document is not an exhaustive study of the various methods, but hopefully serves as an introduction to various technologies and serves as a jumping-off point.

The following networking options are sorted alphabetically - the order does not imply any preferential status.

Cilium

Cilium is open source software for providing and transparently securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies on L3-L7 using an identity based security model that is decoupled from network addressing.

Contiv

Contiv provides configurable networking (native l3 using BGP, overlay using vxlan, classic l2, or Cisco-SDN/ACI) for various use cases. Contiv is all open sourced.

Contrail

Contrail, based on OpenContrail, is a truly open, multi-cloud network virtualization and policy management platform. Contrail / OpenContrail is integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provides different isolation modes for virtual machines, containers/pods and bare metal workloads.

Flannel

Flannel is a very simple overlay network that satisfies the Kubernetes requirements. Many people have reported success with Flannel and Kubernetes.

Google Compute Engine (GCE)

For the Google Compute Engine cluster configuration scripts, we use advanced routing to assign each VM a subnet (default is `/24` - 254 IPs). Any traffic bound for that subnet will be routed directly to the VM by the GCE network fabric. This is in addition to the "main" IP address assigned to the VM, which is NAT'ed for outbound internet access. A linux bridge (called `cbr0`) is configured to exist on that subnet, and is passed to docker's `--bridge` flag.

We start Docker with:

```
DOCKER_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"
```

This bridge is created by Kubelet (controlled by the `--network-plugin=kubenet` flag) according to the `Node`'s `spec.podCIDR`.

Docker will now allocate IPs from the `cbr-cidr` block. Containers can reach each other and `Nodes` over the `cbr0` bridge. Those IPs are all routable within the GCE project network.

GCE itself does not know anything about these IPs, though, so it will not NAT them for outbound internet traffic. To achieve that we use an iptables rule to masquerade (aka SNAT - to make it seem as if packets came from the `Node` itself) traffic that is bound for IPs outside the GCE project network (10.0.0.0/8).

```
iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j MASQUERADE
```

Lastly we enable IP forwarding in the kernel (so the kernel will process packets for bridged containers):

```
sysctl net.ipv4.ip_forward=1
```

The result of all this is that all `Pods` can reach each other and can egress traffic to the internet.

Kube-router

Kube-router is a purpose-built networking solution for Kubernetes that aims to provide high performance and operational simplicity. Kube-router provides a Linux LVS/IPVS-based service proxy, a Linux kernel forwarding-based pod-to-pod networking solution with no overlays, and iptables/ipset-based network policy enforcer.

L2 networks and linux bridging

If you have a “dumb” L2 network, such as a simple switch in a “bare-metal” environment, you should be able to do something similar to the above GCE setup. Note that these instructions have only been tried very casually - it seems to work, but has not been thoroughly tested. If you use this technique and perfect the process, please let us know.

Follow the “With Linux Bridge devices” section of this very nice tutorial from Lars Kellogg-Stedman.

Nuage Networks VCS (Virtualized Cloud Services)

Nuage provides a highly scalable policy-based Software-Defined Networking (SDN) platform. Nuage uses the open source Open vSwitch for the data plane along with a feature rich SDN Controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes Pods and non-Kubernetes environments (VMs and bare metal servers). Nuage’s policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications.The platform’s real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

OpenVSwitch

OpenVSwitch is a somewhat more mature but also complicated way to build an overlay network. This is endorsed by several of the “Big Shops” for networking.

OVN (Open Virtual Networking)

OVN is an opensource network virtualization solution developed by the Open vSwitch community. It lets one create logical switches, logical routers, stateful ACLs, load-balancers etc to build different virtual networking topologies. The project has a specific Kubernetes plugin and documentation at ovn-kubernetes.

Project Calico

Project Calico is an open source container networking provider and network policy engine.

Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet. Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking. Calico also provides fine-grained, intent based network security policy for Kubernetes pods via its distributed firewall.

Calico can also be run in policy enforcement mode in conjunction with other networking solutions such as Flannel, aka canal, or native GCE networking.

Romana

Romana is an open source network and security automation solution that lets you deploy Kubernetes without an overlay network. Romana supports Kubernetes Network Policy to provide isolation across network namespaces.

Weave Net from Weaveworks

Weave Net is a resilient and simple to use network for Kubernetes and its hosted applications. Weave Net runs as a CNI plug-in or stand-alone. In either version, it doesn't require any configuration or extra code to run, and in both cases, the network provides one IP address per pod - as is standard for Kubernetes.

CNI-Genie from Huawei

CNI-Genie is a CNI plugin that enables Kubernetes to simultanously have access to different implementations of the Kubernetes network model in runtime. This includes any implementation that runs as a CNI plugin, such as Flannel, Calico, Romana, Weave-net.

CNI-Genie also supports assigning multiple IP addresses to a pod, each from a different CNI plugin.

Other reading

The early design of the networking model and its rationale, and some future plans are described in more detail in the networking design document.

Network Plugins

Disclaimer: Network plugins are in alpha. Its contents will change rapidly.

Network plugins in Kubernetes come in a few flavors:

- CNI plugins: adhere to the appc/CNI specification, designed for interoperability.
- Kubenet plugin: implements basic `cbr0` using the `bridge` and `host-local` CNI plugins

Installation

The kubelet has a single default network plugin, and a default network common to the entire cluster. It probes for plugins when it starts up, remembers what it found, and executes the selected plugin at appropriate times in the pod lifecycle (this is only true for docker, as rkt manages its own CNI plugins). There are two Kubelet command line parameters to keep in mind when using plugins:

- `network-plugin-dir`: Kubelet probes this directory for plugins on startup
- `network-plugin`: The network plugin to use from `network-plugin-dir`. It must match the name reported by a plugin probed from the plugin directory. For CNI plugins, this is simply "cni".

Network Plugin Requirements

Besides providing the `NetworkPlugin` interface to configure and clean up pod networking, the plugin may also need specific support for kube-proxy. The iptables proxy obviously depends on iptables, and the plugin may need to ensure that container traffic is made available to iptables. For example, if the plugin connects containers to a Linux bridge, the plugin must set the `net/bridge/bridge-nf-call-iptables` sysctl to `1` to ensure that the iptables proxy functions correctly. If the plugin does not use a Linux bridge (but instead something like Open vSwitch or some other mechanism) it should ensure container traffic is appropriately routed for the proxy.

By default if no kubelet network plugin is specified, the `noop` plugin is used, which sets `net/bridge/bridge-nf-call-iptables=1` to ensure simple configurations (like docker with a bridge) work correctly with the iptables proxy.

CNI

The CNI plugin is selected by passing Kubelet the `--network-plugin=cni` command-line option. Kubelet reads a file from `--cni-conf-dir` (default `/etc/cni/net.d`) and uses the CNI configuration from that file to set up each pod's network. The CNI configuration file must match the CNI specification, and any required CNI plugins referenced by the configuration must be present in `--cni-bin-dir` (default `/opt/cni/bin`).

If there are multiple CNI configuration files in the directory, the first one in lexicographic order of file name is used.

In addition to the CNI plugin specified by the configuration file, Kubernetes requires the standard CNI `lo` plugin, at minimum version 0.2.0

Limitation: Due to #31307, `HostPort` won't work with CNI networking plugin at the moment. That means all `hostPort` attribute in pod would be simply ignored.

kubenet

Kubenet is a very basic, simple network plugin, on Linux only. It does not, of itself, implement more advanced features like cross-node networking or network policy. It is typically used together with a cloud provider that sets up routing rules for communication between nodes, or in single-node environments.

Kubenet creates a Linux bridge named `cbr0` and creates a veth pair for each pod with the host end of each pair connected to `cbr0`. The pod end of the pair is assigned an IP address allocated from a range assigned to the node either through configuration or by the controller-manager. `cbr0` is assigned an MTU matching the smallest MTU of an enabled normal interface on the host.

The plugin requires a few things:

- The standard CNI `bridge`, `lo` and `host-local` plugins are required, at minimum version 0.2.0. Kubenet will first search for them in `/opt/cni/bin`. Specify `network-plugin-dir` to supply additional search path. The first found match will take effect.
- Kubelet must be run with the `--network-plugin=kubenet` argument to enable the plugin
- Kubelet should also be run with the `--non-masquerade-cidr=<clusterCidr>` argument to ensure traffic to IPs outside this range will use IP masquerade.
- The node must be assigned an IP subnet through either the `--pod-cidr` kubelet command-line option or the `--allocate-node-cidrs=true --cluster-cidr=<cidr>` controller-manager command-line options.

Customizing the MTU (with kubenet)

The MTU should always be configured correctly to get the best networking performance. Network plugins will usually try to infer a sensible MTU, but sometimes the logic will not result in an optimal MTU. For example, if the Docker bridge or another interface has a small MTU, kubenet will currently select that MTU. Or if you are using IPSEC encapsulation, the MTU must be reduced, and this calculation is out-of-scope for most network plugins.

Where needed, you can specify the MTU explicitly with the `network-plugin-mtu` kubelet option. For example, on AWS the `eth0` MTU is typically 9001, so you might specify `--network-plugin-mtu=9001`. If you're using IPSEC you might reduce it to allow for encapsulation overhead e.g. `--network-plugin-mtu=8873`.

This option is provided to the network-plugin; currently **only kubenet supports** `network-plugin-mtu`.

Usage Summary

- `--network-plugin=cni` specifies that we use the `cni` network plugin with actual CNI plugin binaries located in `--cni-bin-dir` (default `/opt/cni/bin`) and CNI plugin configuration located in `--cni-conf-dir` (default `/etc/cni/net.d`).
- `--network-plugin=kubenet` specifies that we use the `kubenet` network plugin with CNI `bridge` and `host-local` plugins placed in `/opt/cni/bin` or `network-plugin-dir`.
- `--network-plugin-mtu=9001` specifies the MTU to use, currently only used by the `kubenet` network plugin.

Logging Architecture

Application and systems logs can help you understand what is happening inside your cluster. The logs are particularly useful for debugging problems and monitoring cluster activity. Most modern applications have some kind of logging mechanism; as such, most container engines are likewise designed to support some kind of logging. The easiest and most embraced logging method for containerized applications is to write to the standard output and standard error streams.

However, the native functionality provided by a container engine or runtime is usually not enough for a complete logging solution. For example, if a container crashes, a pod is evicted, or a node dies, you'll usually still want to access your application's logs. As such, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called *cluster-level-logging*. Cluster-level logging requires a separate backend to store, analyze, and query logs. Kubernetes provides no native storage solution for log data, but you can integrate many existing logging solutions into your Kubernetes cluster.

Cluster-level logging architectures are described in assumption that a logging backend is present inside or outside of your cluster. If you're not interested in having cluster-level logging, you might still find the description of how logs are stored and handled on the node to be useful.

Basic logging in Kubernetes

In this section, you can see an example of basic logging in Kubernetes that outputs data to the standard output stream. This demonstration uses a pod specification with a container that writes some text to standard output once per second.

counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

To run this pod, use the following command:

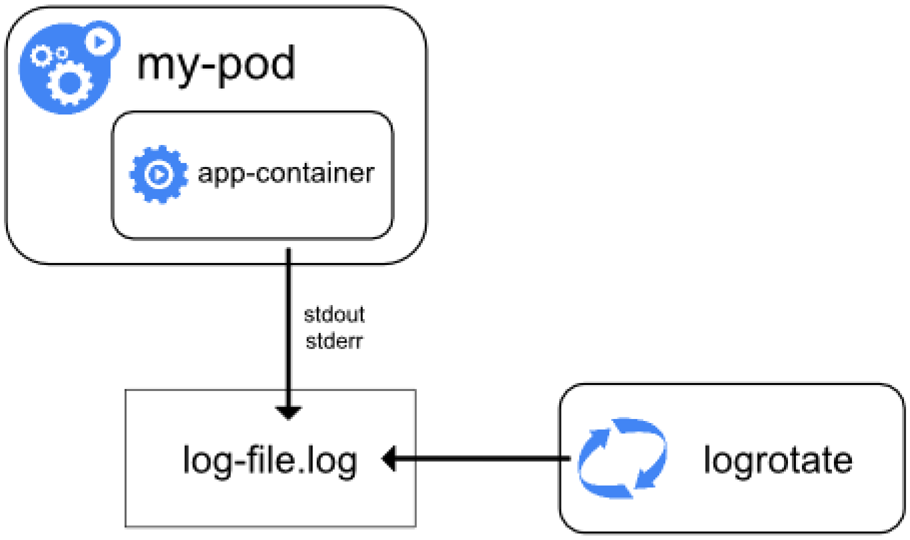
```
$ kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/counter-po
pod "counter" created
```

To fetch the logs, use the `kubectl logs` command, as follows:

```
$ kubectl logs counter
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

You can use `kubectl logs` to retrieve logs from a previous instantiation of a container with `--previous` flag, in case the container has crashed. If your pod has multiple containers, you should specify which container's logs you want to access by appending a container name to the command. See the `kubectl logs` documentation for more details.

Logging at the node level



Everything a containerized application writes to `stdout` and `stderr` is handled and redirected somewhere by a container engine. For example, the Docker container engine redirects those two streams to a logging driver, which is configured in Kubernetes to write to a file in json format.

Note: The Docker json logging driver treats each line as a separate message. When using the Docker logging driver, there is no direct support for multi-line messages. You need to handle multi-line messages at the logging agent level or higher.

By default, if a container restarts, the kubelet keeps one terminated container with its logs. If a pod is evicted from the node, all corresponding containers are also evicted, along with their logs.

An important consideration in node-level logging is implementing log rotation, so that logs don't consume all available storage on the node. Kubernetes currently is not responsible for rotating logs, but rather a deployment tool should set up a solution to address that. For example, in Kubernetes clusters, deployed by the `kube-up.sh` script, there is a `logrotate` tool configured to run each hour. You can also set up a container runtime to rotate application's logs automatically, e.g. by using Docker's `log-opt`. In the `kube-up.sh` script, the latter approach is used for COS image on GCP, and the former approach is used in any other environment. In both cases, by default rotation is configured to take place when log file exceeds 10MB.

As an example, you can find detailed information about how `kube-up.sh` sets up logging for COS image on GCP in the corresponding script.

When you run `kubectl logs` as in the basic logging example, the kubelet on the node handles the request and reads directly from the log file, returning the contents in the response. **Note:** currently, if some external system has performed the rotation, only the contents of the latest log file will be available through `kubectl logs`. E.g. if there's a 10MB file, `logrotate` performs the rotation and there are two files, one 10MB in size and one empty, `kubectl logs` will return an empty response.

System component logs

There are two types of system components: those that run in a container and those that do not run in a container. For example:

- The Kubernetes scheduler and kube-proxy run in a container.
- The kubelet and container runtime, for example Docker, do not run in containers.

On machines with systemd, the kubelet and container runtime write to journald. If systemd is not present, they write to `.log` files in the `/var/log` directory. System components inside containers always write to the `/var/log` directory, bypassing the default logging mechanism. They use the glog logging library. You can find the conventions for logging severity for those components in the development docs on logging.

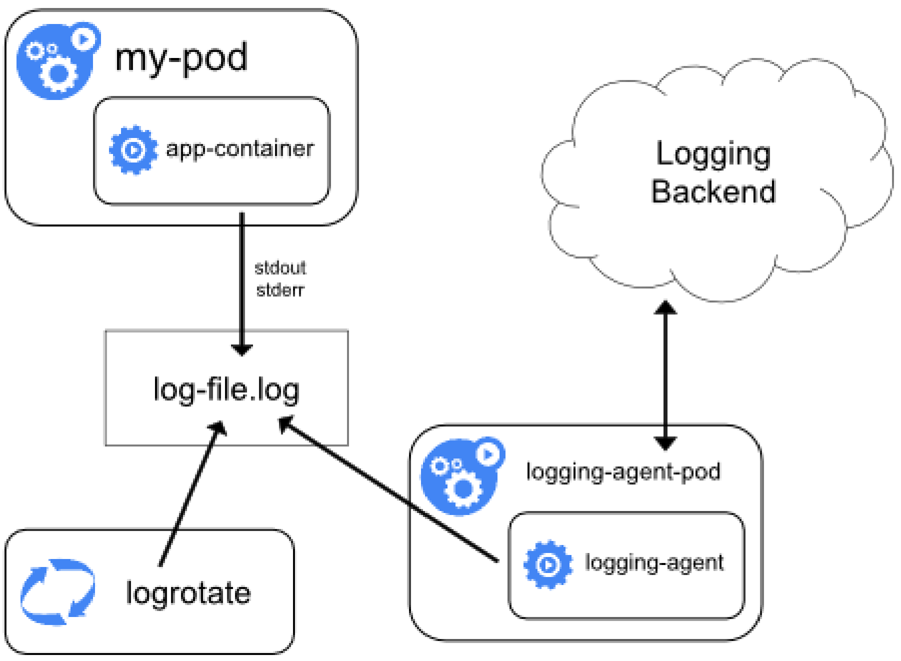
Similarly to the container logs, system component logs in the `/var/log` directory should be rotated. In Kubernetes clusters brought up by the `kube-up.sh` script, those logs are configured to be rotated by the `logrotate` tool daily or once the size exceeds 100MB.

Cluster-level logging architectures

While Kubernetes does not provide a native solution for cluster-level logging, there are several common approaches you can consider. Here are some options:

- Use a node-level logging agent that runs on every node.
- Include a dedicated sidecar container for logging in an application pod.
- Push logs directly to a backend from within an application.

Using a node logging agent



You can implement cluster-level logging by including a *node-level logging agent* on each node. The logging agent is a dedicated tool that exposes logs or pushes logs to a backend. Commonly, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it's common to implement it as either a DaemonSet replica, a manifest pod, or a dedicated native process on the node. However the latter two approaches are deprecated and highly discouraged.

Using a node-level logging agent is the most common and encouraged approach for a Kubernetes cluster, because it creates only one agent per node, and it doesn't require any changes to the applications running on the node. However, node-level logging *only works for applications' standard output and standard error*.

Kubernetes doesn't specify a logging agent, but two optional logging agents are packaged with the Kubernetes release: Stackdriver Logging for use with Google Cloud Platform, and Elasticsearch. You can find more information and instructions in the dedicated documents. Both use fluentd with custom configuration as an agent on the node.

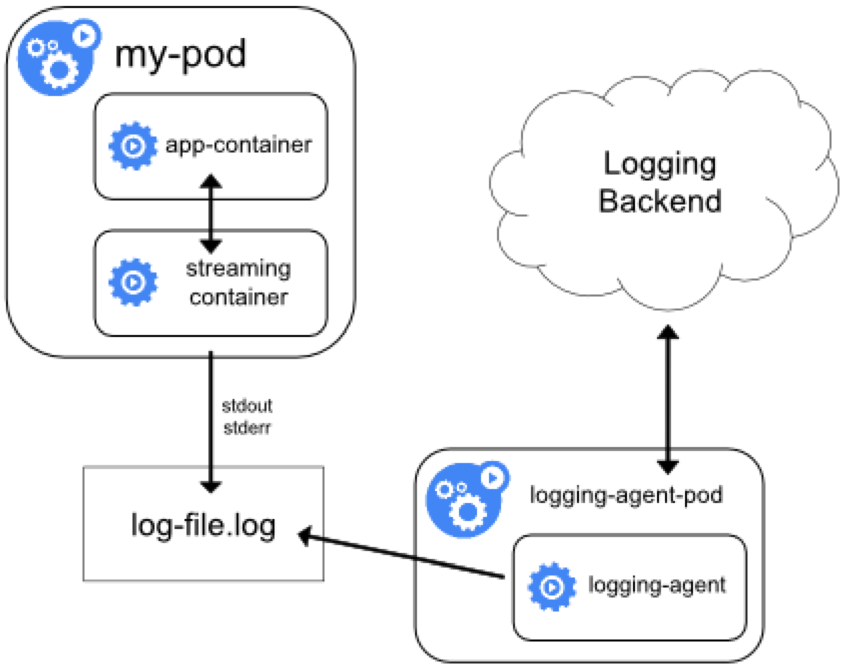
Using a sidecar container with the logging agent

You can use a sidecar container in one of the following ways:

- The sidecar container streams application logs to its own `stdout`.

- The sidecar container runs a logging agent, which is configured to pick up logs from an application container.

Streaming sidecar container



By having your sidecar containers stream to their own `stdout` and `stderr` streams, you can take advantage of the kubelet and the logging agent that already run on each node. The sidecar containers read logs from a file, a socket, or the journald. Each individual sidecar container prints log to its own `stdout` or `stderr` stream.

This approach allows you to separate several log streams from different parts of your application, some of which can lack support for writing to `stdout` or `stderr`. The logic behind redirecting logs is minimal, so it's hardly a significant overhead. Additionally, because `stdout` and `stderr` are handled by the kubelet, you can use built-in tools like `kubectl logs`.

Consider the following example. A pod runs a single container, and the container writes to two different log files, using two different formats. Here's a configuration file for the Pod:

two-files-counter-pod.yaml

two-files-counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

It would be a mess to have log entries of different formats in the same log stream, even if you managed to redirect both components to the `stdout` stream of the container. Instead, you could introduce two sidecar containers. Each sidecar container could tail a particular log file from a shared volume and then redirect the logs to its own `stdout` stream.

Here's a configuration file for a pod that has two sidecar containers:

two-files-counter-pod-streaming-sidecar.yaml two-files-counter-pod-streaming-sidecar.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

Now when you run this pod, you can access each log stream separately by running the following commands:

```
$ kubectl logs counter count-log-1
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

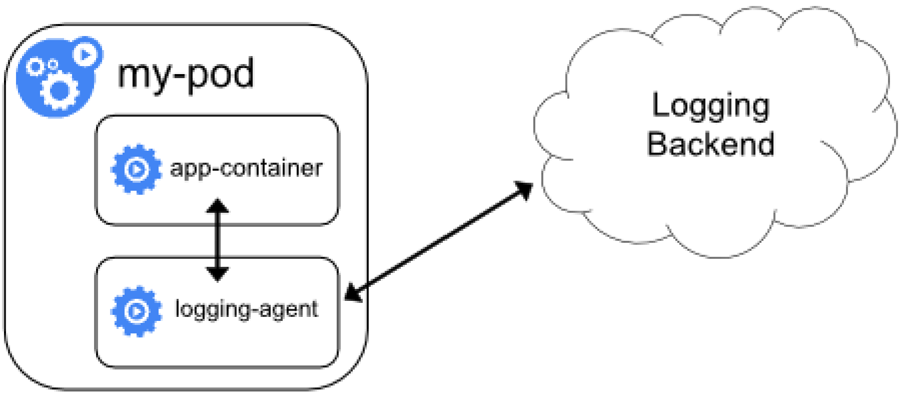
```
$ kubectl logs counter count-log-2
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

The node-level agent installed in your cluster picks up those log streams automatically without any further configuration. If you like, you can configure the agent to parse log lines depending on the source container.

Note, that despite low CPU and memory usage (order of couple of millicores for cpu and order of several megabytes for memory), writing logs to a file and then streaming them to `stdout` can double disk usage. If you have an application that writes to a single file, it's generally better to set `/dev/stdout` as destination rather than implementing the streaming sidecar container approach.

Sidecar containers can also be used to rotate log files that cannot be rotated by the application itself. An example of this approach is a small container running logrotate periodically. However, it's recommended to use `stdout` and `stderr` directly and leave rotation and retention policies to the kubelet.

Sidecar container with a logging agent



If the node-level logging agent is not flexible enough for your situation, you can create a sidecar container with a separate logging agent that you have configured specifically to run with your application.

Note: Using a logging agent in a sidecar container can lead to significant resource consumption. Moreover, you won't be able to access those logs using `kubectl logs` command, because they are not controlled by the kubelet.

As an example, you could use Stackdriver, which uses fluentd as a logging agent. Here are two configuration files that you can use to implement this approach. The first file contains a ConfigMap to configure fluentd.

fluentd-sidecar-config.yaml

```
apiVersion: v1
data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
      format none
      path /var/log/2.log
      pos_file /var/log/2.log.pos
      tag count.format2
    </source>

    <match **>
      type google_cloud
    </match>
kind: ConfigMap
metadata:
  name: fluentd-config
```

Note: The configuration of fluentd is beyond the scope of this article. For information about configuring fluentd, see the official fluentd documentation.

The second file describes a pod that has a sidecar container running fluentd. The pod mounts a volume where fluentd can pick up its configuration data.

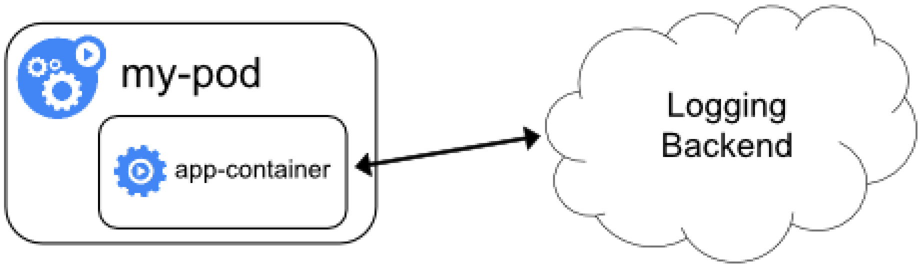
two-files-counter-pod-agent-sidecar.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-agent
    image: gcr.io/google_containers/fluentd-gcp:1.30
    env:
    - name: FLUENTD_ARGS
      value: -c /etc/fluentd-config/fluentd.conf
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-volume
      mountPath: /etc/fluentd-config
  volumes:
  - name: varlog
    emptyDir: {}
  - name: config-volume
    configMap:
      name: fluentd-config
```

After some time you can find log messages in the Stackdriver interface.

Remember, that this is just an example and you can actually replace fluentd with any logging agent, reading from any source inside an application container.

Exposing logs directly from the application



You can implement cluster-level logging by exposing or pushing logs directly from every application; however, the implementation for such a logging mechanism is outside the scope of Kubernetes.

Configuring kubelet Garbage Collection

Garbage collection is a helpful function of kubelet that will clean up unused images and unused containers. Kubelet will perform garbage collection for containers every minute and garbage collection for images every five minutes.

External garbage collection tools are not recommended as these tools can potentially break the behavior of kubelet by removing containers expected to exist.

Image Collection

Kubernetes manages lifecycle of all images through imageManager, with the cooperation of cadvisor.

The policy for garbage collecting images takes two factors into consideration: `HighThresholdPercent` and `LowThresholdPercent`. Disk usage above the high threshold will trigger garbage collection. The garbage collection will delete least recently used images until the low threshold has been met.

Container Collection

The policy for garbage collecting containers considers three user-defined variables. `MinAge` is the minimum age at which a container can be garbage collected. `MaxPerPodContainer` is the maximum number of dead containers every single pod (UID, container name) pair is allowed to have. `MaxContainers` is the maximum number of total dead containers. These variables can be individually disabled by setting `MinAge` to zero and setting `MaxPerPodContainer` and `MaxContainers` respectively to less than zero.

Kubelet will act on containers that are unidentified, deleted, or outside of the boundaries set by the previously mentioned flags. The oldest containers will generally be removed first.

`MaxPerPodContainer` and `MaxContainer` may potentially conflict with each other in situations where retaining the maximum number of containers per pod (`MaxPerPodContainer`) would go outside the allowable range of global dead containers (`MaxContainers`). `MaxPerPodContainer` would be adjusted in this situation: A worst case scenario would be to downgrade `MaxPerPodContainer` to 1 and evict the oldest containers. Additionally, containers owned by pods that have been deleted are removed once they are older than `MinAge` .

Containers that are not managed by kubelet are not subject to container garbage collection.

User Configuration

Users can adjust the following thresholds to tune image garbage collection with the following kubelet flags :

1. `image-gc-high-threshold` , the percent of disk usage which triggers image garbage collection. Default is 90%.
2. `image-gc-low-threshold` , the percent of disk usage to which image garbage collection attempts to free. Default is 80%.

We also allow users to customize garbage collection policy through the following kubelet flags:

1. `minimum-container-ttl-duration` , minimum age for a finished container before it is garbage collected. Default is 0 minute, which means every finished container will be garbage collected.
2. `maximum-dead-containers-per-container` , maximum number of old instances to be retained per container. Default is 1.
3. `maximum-dead-containers` , maximum number of old instances of containers to retain globally. Default is -1, which means there is no global limit.

Containers can potentially be garbage collected before their usefulness has expired. These containers can contain logs and other data that can be useful for troubleshooting. A sufficiently large value for `maximum-dead-containers-per-container` is highly recommended to allow at least 1 dead container to be retained per expected container. A larger value for `maximum-dead-containers` is also recommended for a similar reason. See this issue for more details.

Deprecation

Some kubelet Garbage Collection features in this doc will be replaced by kubelet eviction in the future.

Including:

Existing Flag	New Flag	Rationale
`--image-gc-high-threshold`	`--eviction-hard` or `--eviction-soft`	existing eviction signals can trigger image garbage collection
`--image-gc-low-threshold`	`--eviction-minimum-reclaim`	eviction reclaims achieve the same behavior
`--maximum-dead-containers`		deprecated once old logs are stored outside of container's context
`--maximum-dead-containers-per-container`		deprecated once old logs are stored outside of container's context
`--minimum-container-ttl-duration`		deprecated once old logs are stored outside of container's context
`--low-diskspace-threshold-mb`	`--eviction-hard` or `eviction-soft`	eviction generalizes disk thresholds to other resources
`--outofdisk-transition-frequency`	`--eviction-pressure-transition-period`	eviction generalizes disk pressure transition to other resources

See Configuring Out Of Resource Handling for more details.

Federation

This page explains why and how to manage multiple Kubernetes clusters using federation.

Why federation

Federation makes it easy to manage multiple clusters. It does so by providing 2 major building blocks:

- Sync resources across clusters: Federation provides the ability to keep resources in multiple clusters in sync. For example, you can ensure that the same deployment exists in multiple clusters.
- Cross cluster discovery: Federation provides the ability to auto-configure DNS servers and load balancers with backends from all clusters. For example, you can ensure that a global VIP or DNS record can be used to access backends from multiple clusters.

Some other use cases that federation enables are:

- High Availability: By spreading load across clusters and auto configuring DNS servers and load balancers, federation minimises the impact of cluster failure.
- Avoiding provider lock-in: By making it easier to migrate applications across clusters, federation prevents cluster provider lock-in.

Federation is not helpful unless you have multiple clusters. Some of the reasons why you might want multiple clusters are:

- Low latency: Having clusters in multiple regions minimises latency by serving users from the cluster that is closest to them.
- Fault isolation: It might be better to have multiple small clusters rather than a single large cluster for fault isolation (for example: multiple clusters in different availability zones of a cloud provider). See Multi cluster guide for details.
- Scalability: There are scalability limits to a single kubernetes cluster (this should not be the case for most users. For more details: Kubernetes Scaling and Performance Goals).
- Hybrid cloud: You can have multiple clusters on different cloud providers or on-premises data centers.

Caveats

While there are a lot of attractive use cases for federation, there are also some caveats:

- Increased network bandwidth and cost: The federation control plane watches all clusters to ensure that the current state is as expected. This can lead to significant network cost if the clusters are running in different regions on a cloud provider or on different cloud providers.
- Reduced cross cluster isolation: A bug in the federation control plane can impact all clusters. This is mitigated by keeping the logic in federation control plane to a minimum. It mostly delegates to the control plane in kubernetes clusters whenever it can. The design and implementation also errs on the side of safety and avoiding multi-cluster outage.
- Maturity: The federation project is relatively new and is not very mature. Not all resources are available and many are still alpha. Issue 38893 enumerates known issues with the system that the team is busy solving.

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (e.g. Google Cloud, AWS), and on-premises (e.g. on OpenStack). Simply create all of the clusters that you require, in the appropriate cloud providers and/or locations, and register each cluster's API endpoint and credentials with your Federation API Server (See the federation admin guide for details).

Thereafter, your API resources can span different clusters and cloud providers.

Setting up federation

To be able to federate multiple clusters, you first need to set up a federation control plane. Follow the setup guide to set up the federation control plane.

API resources

Once you have the control plane set up, you can start creating federation API resources. The following guides explain some of the resources in detail:

- Cluster
- ConfigMap
- DaemonSets
- Deployment
- Events
- Hpa
- Ingress
- Jobs
- Namespaces
- ReplicaSets
- Secrets
- Services

API reference docs lists all the resources supported by federation apiserver.

Cascading deletion

Kubernetes version 1.6 includes support for cascading deletion of federated resources. With cascading deletion, when you delete a resource from the federation control plane, you also delete the corresponding resources in all underlying clusters.

Cascading deletion is not enabled by default when using the REST API. To enable it, set the option `DeleteOptions.orphanDependents=false` when you delete a resource from the federation control plane using the REST API. Using `kubectl delete` enables cascading deletion by default. You can disable it by running `kubectl delete --cascade=false`

Note: Kubernetes version 1.5 included cascading deletion support for a subset of federation resources.

Scope of a single cluster

On IaaS providers such as Google Compute Engine or Amazon Web Services, a VM exists in a zone or availability zone. We suggest that all the VMs in a Kubernetes cluster should be in the same availability zone, because:

- compared to having a single global Kubernetes cluster, there are fewer single-points of failure.
- compared to a cluster that spans availability zones, it is easier to reason about the availability properties of a single-zone cluster.
- when the Kubernetes developers are designing the system (e.g. making assumptions about latency, bandwidth, or correlated failures) they are assuming all the machines are in a single data center, or otherwise closely connected.

It is okay to have multiple clusters per availability zone, though on balance we think fewer is better. Reasons to prefer fewer clusters are:

- improved bin packing of Pods in some cases with more nodes in one cluster (less resource fragmentation).
- reduced operational overhead (though the advantage is diminished as ops tooling and processes mature).
- reduced costs for per-cluster fixed resource costs, e.g. apiserver VMs (but small as a percentage of overall cluster cost for medium to large clusters).

Reasons to have multiple clusters include:

- strict security policies requiring isolation of one class of work from another (but, see Partitioning Clusters below).
- test clusters to canary new Kubernetes releases or other cluster software.

Selecting the right number of clusters

The selection of the number of Kubernetes clusters may be a relatively static choice, only revisited occasionally. By contrast, the number of nodes in a cluster and the number of pods in a service may change frequently according to load and growth.

To pick the number of clusters, first, decide which regions you need to be in to have adequate latency to all your end users, for services that will run on Kubernetes (if you use a Content Distribution Network, the latency requirements for the CDN-hosted content need not be considered). Legal issues might influence this as well. For example, a company with a global customer base might decide to have clusters in US, EU, AP, and SA regions. Call the number of regions to be in `R`.

Second, decide how many clusters should be able to be unavailable at the same time, while still being available. Call the number that can be unavailable `U`. If you are not sure, then 1 is a fine choice.

If it is allowable for load-balancing to direct traffic to any region in the event of a cluster failure, then you need at least the larger of `R` or `U + 1` clusters. If it is not (e.g. you want to ensure low latency for all users in the event of a cluster failure), then you need to have `R * (U + 1)` clusters (`U + 1` in each of `R` regions). In any case, try to put each cluster in a different zone.

Finally, if any of your clusters would need more than the maximum recommended number of nodes for a Kubernetes cluster, then you may need even more clusters. Kubernetes v1.3 supports clusters up to 1000 nodes in size.

What's next

- Learn more about the Federation proposal.
- See this setup guide for cluster federation.

- See this Kubecon2016 talk on federation

Using Sysctls in a Kubernetes Cluster

This document describes how sysctls are used within a Kubernetes cluster.

What is a Sysctl?

In Linux, the sysctl interface allows an administrator to modify kernel parameters at runtime. Parameters are available via the `/proc/sys/` virtual process file system. The parameters cover various subsystems such as:

- kernel (common prefix: `kernel.`)
- networking (common prefix: `net.`)
- virtual memory (common prefix: `vm.`)
- MDADM (common prefix: `dev.`)
- More subsystems are described in Kernel docs.

To get a list of all parameters, you can run

```
$ sudo sysctl -a
```

Namespaced vs. Node-Level Sysctls

A number of sysctls are *namespaced* in today's Linux kernels. This means that they can be set independently for each pod on a node. Being namespaced is a requirement for sysctls to be accessible in a pod context within Kubernetes.

The following sysctls are known to be *namespaced*:

- `kernel.shm*`,
- `kernel.msg*`,
- `kernel.sem`,
- `fs.mqueue.*`,
- `net.*`.

Sysctls which are not namespaced are called *node-level* and must be set manually by the cluster admin, either by means of the underlying Linux distribution of the nodes (e.g. via `/etc/sysctls.conf`) or using a DaemonSet with privileged containers.

Note: it is good practice to consider nodes with special sysctl settings as *tainted* within a cluster, and only schedule pods onto them which need those sysctl settings. It is suggested to use the Kubernetes *taints and toleration* feature to implement this.

Safe vs. Unsafe Sysctls

Sysctls are grouped into *safe* and *unsafe* sysctls. In addition to proper namespacing a *safe* sysctl must be properly *isolated* between pods on the same node. This means that setting a *safe* sysctl for one pod

- must not have any influence on any other pod on the node
- must not allow to harm the node's health
- must not allow to gain CPU or memory resources outside of the resource limits of a pod.

By far, most of the *namespaced* sysctls are not necessarily considered *safe*.

For Kubernetes 1.4, the following sysctls are supported in the *safe* set:

- `kernel.shm_rmid_forced`,
- `net.ipv4.ip_local_port_range`,
- `net.ipv4.tcp_syncookies`.

This list will be extended in future Kubernetes versions when the kubelet supports better isolation mechanisms.

All *safe* sysctls are enabled by default.

All *unsafe* sysctls are disabled by default and must be allowed manually by the cluster admin on a per-node basis. Pods with disabled unsafe sysctls will be scheduled, but will fail to launch.

Warning: Due to their nature of being *unsafe*, the use of *unsafe* sysctls is at-your-own-risk and can lead to severe problems like wrong behavior of containers, resource shortage or complete breakage of a node.

Enabling Unsafe Sysctls

With the warning above in mind, the cluster admin can allow certain *unsafe* sysctls for very special situations like e.g. high-performance or real-time application tuning. *Unsafe* sysctls are enabled on a node-by-node basis with a flag of the kubelet, e.g.:

```
$ kubelet --experimental-allowed-unsafe-sysctls 'kernel.msg*,net.ipv4.route.min_pm
```

Only *namespaced* sysctls can be enabled this way.

Setting Sysctls for a Pod

The sysctl feature is an alpha API in Kubernetes 1.4. Therefore, sysctls are set using annotations on pods. They apply to all containers in the same pod.

Here is an example, with different annotations for *safe* and *unsafe* sysctls:

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  annotations:
    security.alpha.kubernetes.io/sysctls: kernel.shm_rmid_forced=1
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.min_pmtu=1000,kern
spec:
  ...
```

Note: a pod with the *unsafe* sysctls specified above will fail to launch on any node which has not enabled those two *unsafe* sysctls explicitly. As with *node-level* sysctls it is recommended to use *taints and toleration* feature or taints on nodes to schedule those pods onto the right nodes.

Proxies in Kubernetes

This page explains proxies used with Kubernetes.

Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The kubectl proxy: - runs on a user's desktop or in a pod - proxies from a localhost address to the Kubernetes apiserver - client to proxy uses HTTP - proxy to apiserver uses HTTPS - locates apiserver - adds authentication headers
2. The apiserver proxy: - is a bastion built into the apiserver - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable - runs in the apiserver processes - client to proxy uses HTTPS (or http if apiserver so configured) - proxy to target may use HTTP or HTTPS as chosen by proxy using available information - can be used to reach a Node, Pod, or Service - does load balancing when used to reach a Service
3. The kube proxy: - runs on each node - proxies UDP and TCP - does not understand HTTP - provides load balancing - is just used to reach services
4. A Proxy/Load-balancer in front of apiserver(s): - existence and implementation varies from cluster to cluster (e.g. nginx) - sits between all clients and one or more apiservers - acts as load balancer if there are several apiservers.
5. Cloud Load Balancers on external services: - are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer) - are created automatically when the Kubernetes service has type `LoadBalancer` - use UDP/TCP only - implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

Requesting redirects

Proxies have replaced redirect capabilities. Redirects have been deprecated.

Controller manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager.

What are controller manager metrics

Controller manager metrics provide important insight into the performance and health of the controller manager. These metrics include common Go language runtime metrics such as go_routine count and controller specific metrics such as etcd request latencies or Cloudprovider (AWS, GCE, Openstack) API latencies that can be used to gauge the health of a cluster.

Starting from Kubernetes 1.7, detailed Cloudprovider metrics are available for storage operations for GCE, AWS, Vsphere and Openstack. These metrics can be used to monitor health of persistent volume operations.

For example, for GCE these metrics are called:

```
cloudprovider_gce_api_request_duration_seconds { request = "instance_list"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_insert"}
cloudprovider_gce_api_request_duration_seconds { request = "disk_delete"}
cloudprovider_gce_api_request_duration_seconds { request = "attach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "detach_disk"}
cloudprovider_gce_api_request_duration_seconds { request = "list_disk"}
```

Configuration

In a cluster, controller-manager metrics are available from `http://localhost:10252/metrics` from the host where the controller-manager is running.

The metrics are emitted in prometheus format and are human readable.

In a production environment you may want to configure prometheus or some other metrics scraper to periodically gather these metrics and make them available in some kind of time series database.

Device Plugins

FEATURE STATE: `Kubernetes v1.8` alpha

Starting in version 1.8, Kubernetes provides a device plugin framework for vendors to advertise their resources to the kubelet without changing Kubernetes core code. Instead of writing custom Kubernetes code, vendors can implement a device plugin that can be deployed manually or as a DaemonSet. The targeted devices include GPUs, High-performance NICs, FPGAs, InfiniBand, and other similar computing resources that may require vendor specific initialization and setup.

Device plugin registration

The device plugins feature is gated by the `DevicePlugins` feature gate and is disabled by default. When the device plugins feature is enabled, the kubelet exports a `Registration` gRPC service:

```
service Registration {
        rpc Register(RegisterRequest) returns (Empty) {}
}
```

A device plugin can register itself with the kubelet through this gRPC service. During the registration, the device plugin needs to send:

- The name of its Unix socket.
- The Device Plugin API version against which it was built.
- The `ResourceName` it wants to advertise. Here `ResourceName` needs to follow the extended resource naming scheme as `vendor-domain/resource` . For example, an Nvidia GPU is advertised as `nvidia.com/gpu` .

Following a successful registration, the device plugin sends the kubelet the list of devices it manages, and the kubelet is then in charge of advertising those resources to the API server as part of the kubelet node status update. For example, after a device plugin registers `vendor-domain/foo` with the kubelet and reports two healthy devices on a node, the node status is updated to advertise 2 `vendor-domain/foo`.

Then, developers can request devices in a Container specification by using the same process that is used for opaque integer resources. In version 1.8, extended resources are spported only as integer resources and must have `limit` equal to `request` in the Container specification.

Device plugin implementation

The general workflow of a device plugin includes the following steps:

- Initialization. During this phase, the device plugin performs vendor specific initialization and setup to make sure the devices are in a ready state.
- The plugin starts a gRPC service, with a Unix socket under host path `/var/lib/kubelet/device-plugins/`, that implements the following interfaces:

```
service DevicePlugin {
      // ListAndWatch returns a stream of List of Devices
      // Whenever a Device state change or a Device disapears, ListAndWatch
      // returns the new list
      rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

      // Allocate is called during container creation so that the Device
      // Plugin can run device specific operations and instruct Kubelet
      // of the steps to make the Device available in the container
      rpc Allocate(AllocateRequest) returns (AllocateResponse) {}
}
```

- The plugin registers itself with the kubelet through the Unix socket at host path `/var/lib/kubelet/device-plugins/kubelet.sock`.

- After successfully registering itself, the device plugin runs in serving mode, during which it keeps monitoring device health and reports back to the kubelet upon any device state changes. It is also responsible for serving `Allocate` gRPC requests. During `Allocate`, the device plugin may do device-specific preparation; for example, GPU cleanup or QRNG initialization. If the operations succeed, the device plugin returns an `AllocateResponse` that contains container runtime configurations for accessing the allocated devices. The kubelet passes this information to the container runtime.

A device plugin is expected to detect kubelet restarts and re-register itself with the new kubelet instance. In version 1.8, a new kubelet instance cleans up all the existing Unix sockets under `/var/lib/kubelet/device-plugins` when it starts. A device plugin can monitor the deletion of its Unix socket and re-register itself upon such an event.

Device plugin deployment

A device plugin can be deployed manually or as a DaemonSet. Being deployed as a DaemonSet has the benefit that Kubernetes can restart the device plugin if it fails. Otherwise, an extra mechanism is needed to recover from device plugin failures. The canonical directory `/var/lib/kubelet/device-plugins` requires privileged access, so a device plugin must run in a privileged security context. If a device plugin is running as a DaemonSet, `/var/lib/kubelet/device-plugins` must be mounted as a Volume in the plugin's PodSpec.

Examples

For an example device plugin implementation, see nvidia GPU device plugin for COS base OS.

Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a `ResourceQuota` object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.

Resource quotas work like this:

- Different teams work in different namespaces. Currently this is voluntary, but support for making this mandatory via ACLs is planned.
- The administrator creates one or more Resource Quota objects for each namespace.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a Resource Quota.
- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code `403 FORBIDDEN` with a message explaining the constraint that would have been violated.
- If quota is enabled in a namespace for compute resources like `cpu` and `memory`, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation. Hint: Use the LimitRange admission controller to force defaults for pods that make no compute resource requirements. See the walkthrough for an example to avoid this problem.

Examples of policies that could be created using namespaces and quotas are:

- In a cluster with a capacity of 32 GiB RAM, and 16 cores, let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores, and hold 2GiB and 2 cores in reserve for future allocation.

- Limit the "testing" namespace to using 1 core and 1GiB RAM. Let the "production" namespace use any amount.

In the case where the total capacity of the cluster is less than the sum of the quotas of the namespaces, there may be contention for resources. This is handled on a first-come-first-served basis.

Neither contention nor changes to quota will affect already created resources.

Enabling Resource Quota

Resource Quota support is enabled by default for many Kubernetes distributions. It is enabled when the apiserver `--admission-control=` flag has `ResourceQuota` as one of its arguments.

Resource Quota is enforced in a particular namespace when there is a `ResourceQuota` object in that namespace. There should be at most one `ResourceQuota` object in a namespace.

Compute Resource Quota

You can limit the total sum of compute resources that can be requested in a given namespace.

The following resource types are supported:

Resource Name	Description
`cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
`limits.cpu`	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
`limits.memory`	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
`memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.

Storage Resource Quota

You can limit the total sum of storage resources that can be requested in a given namespace.

In addition, you can limit consumption of storage resources based on associated storage-class.

Resource Name	Description
`requests.storage`	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the namespace.
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	Across all persistent volume claims associated with the storage-class-name, the sum of storage requests cannot exceed this value.
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	Across all persistent volume claims associated with the storage-class-name, the total number of persistent volume claims that can exist in the namespace.

For example, if an operator wants to quota storage with `gold` storage class separate from `bronze` storage class, the operator can define a quota as follows:

- `gold.storageclass.storage.k8s.io/requests.storage: 500Gi`
- `bronze.storageclass.storage.k8s.io/requests.storage: 100Gi`

In release 1.8, quota support for local ephemeral storage is added as alpha feature

Resource Name	Description
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.

Object Count Quota

The number of objects of a given type can be restricted. The following types are supported:

Resource Name	Description
`configmaps`	The total number of config maps that can exist in the namespace.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the namespace.
`pods`	The total number of pods in a non-terminal state that can exist in the namespace. A pod is in a terminal state if `status.phase in (Failed, Succeeded)` is true.
`replicationcontrollers`	The total number of replication controllers that can exist in the namespace.
`resourcequotas`	The total number of resource quotas that can exist in the namespace.
`services`	The total number of services that can exist in the namespace.
`services.loadbalancers`	The total number of services of type load balancer that can exist in the namespace.
`services.nodeports`	The total number of services of type node port that can exist in the namespace.
`secrets`	The total number of secrets that can exist in the namespace.

For example, `pods` quota counts and enforces a maximum on the number of `pods` created in a single namespace.

You might want to set a pods quota on a namespace to avoid the case where a user creates many small pods and exhausts the cluster's supply of Pod IPs.

Quota Scopes

Each quota can have an associated set of scopes. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to the quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Scope	Description
`Terminating`	Match pods where `spec.activeDeadlineSeconds >= 0`
`NotTerminating`	Match pods where `spec.activeDeadlineSeconds is nil`

Scope	Description
`BestEffort`	Match pods that have best effort quality of service.
`NotBestEffort`	Match pods that do not have best effort quality of service.

The `BestEffort` scope restricts a quota to tracking the following resource: `pods`

The `Terminating`, `NotTerminating`, and `NotBestEffort` scopes restrict a quota to tracking the following resources:

- `cpu`
- `limits.cpu`
- `limits.memory`
- `memory`
- `pods`
- `requests.cpu`
- `requests.memory`

Requests vs Limits

When allocating compute resources, each container may specify a request and a limit value for either CPU or memory. The quota can be configured to quota either value.

If the quota has a value specified for `requests.cpu` or `requests.memory`, then it requires that every incoming container makes an explicit request for those resources. If the quota has a value specified for `limits.cpu` or `limits.memory`, then it requires that every incoming container specifies an explicit limit for those resources.

Viewing and Setting Quotas

Kubectl supports creating, updating, and viewing quotas:

```
$ kubectl create namespace myspace

$ cat <<EOF > compute-resources.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
EOF
$ kubectl create -f ./compute-resources.yaml --namespace=myspace

$ cat <<EOF > object-counts.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
EOF
$ kubectl create -f ./object-counts.yaml --namespace=myspace

$ kubectl get quota --namespace=myspace
NAME                    AGE
compute-resources       30s
object-counts           32s

$ kubectl describe quota compute-resources --namespace=myspace
Name:                  compute-resources
Namespace:             myspace
Resource               Used Hard
--------               ---- ----
limits.cpu             0    2
limits.memory          0    2Gi
pods                   0    4
requests.cpu           0    1
requests.memory        0    1Gi

$ kubectl describe quota object-counts --namespace=myspace
Name:                   object-counts
```

```
Namespace:               myspace
Resource                 Used    Hard
--------                 ----    ----
configmaps               0       10
persistentvolumeclaims   0       4
replicationcontrollers   0       20
secrets                  1       10
services                 0       10
services.loadbalancers   0       2
```

Quota and Cluster Capacity

Resource Quota objects are independent of the Cluster Capacity. They are expressed in absolute units. So, if you add nodes to your cluster, this does *not* automatically give each namespace the ability to consume more resources.

Sometimes more complex policies may be desired, such as:

- Proportionally divide total cluster resources among several teams.
- Allow each tenant to grow resource usage as needed, but have a generous limit to prevent accidental resource exhaustion.
- Detect demand from one namespace, add nodes, and increase quota.

Such policies could be implemented using ResourceQuota as a building-block, by writing a ‘controller’ which watches the quota usage and adjusts the quota hard limits of each namespace according to other signals.

Note that resource quota divides up aggregate cluster resources, but it creates no restrictions around nodes: pods from several namespaces may run on the same node.

Example

See a detailed example for how to use resource quota.

Read More

See ResourceQuota design doc for more information.

Pod Security Policies

Objects of type `PodSecurityPolicy` govern the ability to make requests on a pod that affect the `SecurityContext` that will be applied to a pod and container.

See PodSecurityPolicy proposal for more information.

What is a Pod Security Policy?

A *Pod Security Policy* is a cluster-level resource that controls the actions that a pod can perform and what it has the ability to access. The `PodSecurityPolicy` objects define a set of conditions that a pod must run with in order to be accepted into the system. They allow an administrator to control the following:

Control Aspect	Field Name
Running of privileged containers	`privileged`

Control Aspect	Field Name
Default set of capabilities that will be added to a container	`defaultAddCapabilities`
Capabilities that will be dropped from a container	`requiredDropCapabilities`
Capabilities a container can request to be added	`allowedCapabilities`
Controlling the usage of volume types	`volumes`
The use of host networking	`hostNetwork`
The use of host ports	`hostPorts`
The use of host's PID namespace	`hostPID`
The use of host's IPC namespace	`hostIPC`
The SELinux context of the container	`seLinux`
The user ID	`runAsUser`
Configuring allowable supplemental groups	`supplementalGroups`
Allocating an FSGroup that owns the pod's volumes	`fsGroup`
Requiring the use of a read only root file system	`readOnlyRootFilesystem`
Running of a container that allow privilege escalation from its parent	`allowPrivilegeEscalation`
Control whether a process can gain more privileges than its parent process	`defaultAllowPrivilegeEscalation`

Pod Security Policies are comprised of settings and strategies that control the security features a pod has access to. These settings fall into three categories:

- *Controlled by a Boolean*: Fields of this type default to the most restrictive value.
- *Controlled by an allowable set*: Fields of this type are checked against the set to ensure their values are allowed.
- *Controlled by a strategy*: Items that have a strategy to provide a mechanism to generate the value and a mechanism to ensure that a specified value falls into the set of allowable values.

Strategies

RunAsUser

- *MustRunAs* - Requires a `range` to be configured. Uses the first value of the range as the default. Validates against the configured range.
- *MustRunAsNonRoot* - Requires that the pod be submitted with a non-zero `runAsUser` or have the `USER` directive defined in the image. No default provided.
- *RunAsAny* - No default provided. Allows any `runAsUser` to be specified.

SELinux

- *MustRunAs* - Requires `seLinuxOptions` to be configured if not using pre-allocated values. Uses `seLinuxOptions` as the default. Validates against `seLinuxOptions`.
- *RunAsAny* - No default provided. Allows any `seLinuxOptions` to be specified.

SupplementalGroups

- *MustRunAs* - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against all ranges.
- *RunAsAny* - No default provided. Allows any `supplementalGroups` to be specified.

FSGroup

- *MustRunAs* - Requires at least one range to be specified. Uses the minimum value of the first range as the default. Validates against the first ID in the first range.
- *RunAsAny* - No default provided. Allows any `fsGroup` ID to be specified.

Controlling Volumes

The usage of specific volume types can be controlled by setting the volumes field of the PSP. The allowable values of this field correspond to the volume sources that are defined when creating a volume:

1. azureFile
2. azureDisk
3. flocker
4. flexVolume
5. hostPath
6. emptyDir
7. gcePersistentDisk
8. awsElasticBlockStore
9. gitRepo
10. secret
11. nfs
12. iscsi
13. glusterfs
14. persistentVolumeClaim
15. rbd
16. cinder
17. cephFS
18. downwardAPI
19. fc
20. configMap
21. vsphereVolume
22. quobyte
23. photonPersistentDisk
24. projected

25. portworxVolume
26. scaleIO
27. storageos
28. * (allow all volumes)

The recommended minimum set of allowed volumes for new PSPs are configMap, downwardAPI, emptyDir, persistentVolumeClaim, secret, and projected.

Host Network

- *HostPorts*, default `empty`. List of `HostPortRange`, defined by `min` (inclusive) and `max` (inclusive), which define the allowed host ports.

AllowPrivilegeEscalation

Gates whether or not a user is allowed to set the security context of a container to `allowPrivilegeEscalation=true`. This field defaults to `false`.

DefaultAllowPrivilegeEscalation

Sets the default for the security context `AllowPrivilegeEscalation` of a container. This bool directly controls whether the `no_new_privs` flag gets set on the container process. It defaults to `nil`. The default behavior of `nil` allows privilege escalation so as to not break setuid binaries. Setting it to `false` ensures that no child process of a container can gain more privileges than its parent.

Admission

Admission control with `PodSecurityPolicy` allows for control over the creation and modification of resources based on the capabilities allowed in the cluster.

Admission uses the following approach to create the final security context for the pod:

1. Retrieve all PSPs available for use.
2. Generate field values for security context settings that were not specified on the request.
3. Validate the final settings against the available policies.

If a matching policy is found, then the pod is accepted. If the request cannot be matched to a PSP, the pod is rejected.

A pod must validate every field against the PSP.

Creating a Pod Security Policy

Here is an example Pod Security Policy. It has permissive settings for all fields

psp.yaml

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - min: 8000
    max: 8080
  volumes:
  - '*'
  allowedCapabilities:
  - '*'
```

Create the policy by downloading the example file and then running this command:

```
$ kubectl create -f ./psp.yaml
podsecuritypolicy "permissive" created
```

Getting a list of Pod Security Policies

To get a list of existing policies, use `kubectl get`:

```
$ kubectl get psp
NAME        PRIV   CAPS  SELINUX   RUNASUSER         FSGROUP   SUPGROUP  READONLYR
permissive  false  []    RunAsAny  RunAsAny          RunAsAny  RunAsAny  false
privileged  true   []    RunAsAny  RunAsAny          RunAsAny  RunAsAny  false
restricted  false  []    RunAsAny  MustRunAsNonRoot  RunAsAny  RunAsAny  false
```

Editing a Pod Security Policy

To modify policy interactively, use `kubectl edit`:

```
$ kubectl edit psp permissive
```

This command will open a default text editor where you will be able to modify policy.

Deleting a Pod Security Policy

Once you don't need a policy anymore, simply delete it with `kubectl`:

```
$ kubectl delete psp permissive
podsecuritypolicy "permissive" deleted
```

Enabling Pod Security Policies

In order to use Pod Security Policies in your cluster you must ensure the following

1. You have enabled the API type `extensions/v1beta1/podsecuritypolicy` (only for versions prior 1.6)
2. You have enabled the admission controller `PodSecurityPolicy`
3. You have defined your policies

Working With RBAC

In Kubernetes 1.5 and newer, you can use PodSecurityPolicy to control access to privileged containers based on user role and groups. Access to different PodSecurityPolicy objects can be controlled via authorization.

Note that Controller Manager must be run against the secured API port, and must not have superuser permissions. Otherwise requests would bypass authentication and authorization modules, all PodSecurityPolicy objects would be allowed, and user will be able to create privileged containers.

PodSecurityPolicy authorization uses the union of all policies available to the user creating the pod and the service account specified on the pod.

Access to given PSP policies for a user will be effective only when creating Pods directly.

For pods created on behalf of a user, in most cases by Controller Manager, access should be given to the service account specified on the pod spec template. Examples of resources that create pods on behalf of a user are Deployments, ReplicaSets, etc.

For more details, see the PodSecurityPolicy RBAC example of applying PodSecurityPolicy to control access to privileged containers based on role and groups when deploying Pods directly.

Tasks

This section of the Kubernetes documentation contains pages that show how to do individual tasks. A task page shows how to do a single thing, typically by giving a short sequence of steps.

Web UI (Dashboard)

Deploy and access the Dashboard web user interface to help you manage and monitor containerized applications in a Kubernetes cluster.

Using the kubectl Command-line

Install and setup the `kubectl` command-line tool used to directly manage Kubernetes clusters.

Configuring Pods and Containers

Perform common configuration tasks for Pods and Containers.

Running Applications

Perform common application management tasks, such as rolling updates, injecting information into pods, and horizontal Pod autoscaling.

Running Jobs

Run Jobs using parallel processing.

Accessing Applications in a Cluster

Configure load balancing, port forwarding, or setup firewall or DNS configurations to access applications in a cluster.

Monitoring, Logging, and Debugging

Setup monitoring and logging to troubleshoot a cluster or debug a containerized application.

Accessing the Kubernetes API

Learn various methods to directly access the Kubernetes API.

Using TLS

Configure your application to trust and use the cluster root Certificate Authority (CA).

Administering a Cluster

Learn common tasks for administering a cluster.

Administering Federation

Configure components in a cluster federation.

Managing Stateful Applications

Perform common tasks for managing Stateful applications, including scaling, deleting, and debugging StatefulSets.

Cluster Daemons

Perform common tasks for managing a DaemonSet, such as performing a rolling update.

Managing GPUs

Configure and schedule NVIDIA GPUs for use as a resource by nodes in a cluster.

Managing HugePages

Configure and schedule huge pages as a schedulable resource in a cluster.

What's next

If you would like to write a task page, see Creating a Documentation Pull Request.

Install and Set Up kubectl

Here are a few methods to install kubectl.

Use the Kubernetes command-line tool, kubectl, to deploy and manage applications on Kubernetes. Using kubectl, you can inspect cluster resources; create, delete, and update components; and look at your new cluster and bring up example apps.

Before you begin

Use a version of kubectl that is the same version as your server or later. Using an older kubectl with a newer server might produce validation errors.

Install kubectl binary via curl

macOS | Linux | Windows

1. Download the latest release with the command:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s
```

To download a specific version, replace the `$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)` portion of the command with the specific version.

For example, to download version v1.8.0 on MacOS, type:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.8.0/b
```

2. Make the kubectl binary executable.

   `chmod +x ./kubectl`
3. Move the binary in to your PATH.

   `sudo mv ./kubectl /usr/local/bin/kubectl`

Download as part of the Google Cloud SDK

kubectl can be installed as part of the Google Cloud SDK.

1. Install the Google Cloud SDK.
2. Run the following command to install `kubectl`:

```
gcloud components install kubectl
```

3. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install with snap on Ubuntu

kubectl is available as a snap application.

1. If you are on Ubuntu or one of other Linux distributions that support snap package manager, you can install with:

   ```
   sudo snap install kubectl --classic
   ```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install with Homebrew on macOS

1. If you are on macOS and using Homebrew package manager, you can install with:

   ```
   brew install kubectl
   ```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install with Chocolatey on Windows

1. If you are on Windows and using Chocolatey package manager, you can install with:

   ```
   choco install kubernetes-cli
   ```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.
3. Configure kubectl to use a remote Kubernetes cluster:

   ```
   cd C:\users\yourusername (Or wherever your %HOME% directory is)
   mkdir .kube
   cd .kube
   touch config
   ```

Edit the config file with a text editor of your choice, such as Notepad for example.

Configure kubectl

In order for kubectl to find and access a Kubernetes cluster, it needs a kubeconfig file, which is created automatically when you create a cluster using kube-up.sh or successfully deploy a Minikube cluster. See the getting started guides for more about creating clusters. If you need access to a cluster you didn't create, see the Sharing Cluster Access document. By default, kubectl configuration is located at `~/.kube/config`.

Check the kubectl configuration

Check that kubectl is properly configured by getting the cluster state:

```
kubectl cluster-info
```

If you see a URL response, kubectl is correctly configured to access your cluster.

If you see a message similar to the following, kubectl is not correctly configured:

```
The connection to the server <server-name:port> was refused - did you specify the
```

Enabling shell autocompletion

kubectl includes autocompletion support, which can save a lot of typing!

The completion script itself is generated by kubectl, so you typically just need to invoke it from your profile.

Common examples are provided here. For more details, consult `kubectl completion -h`.

On Linux, using bash

To add kubectl autocompletion to your current shell, run `source <(kubectl completion bash)`.

To add kubectl autocompletion to your profile, so it is automatically loaded in future shells run:

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

On macOS, using bash

On macOS, you will need to install bash-completion support via Homebrew first:

```
## If running Bash 3.2 included with macOS
brew install bash-completion
## or, if running Bash 4.1+
brew install bash-completion@2
```

Follow the “caveats” section of brew’s output to add the appropriate bash completion path to your local .bashrc.

If you’ve installed kubectl using the Homebrew instructions then kubectl completion should start working immediately.

If you have installed kubectl manually, you need to add kubectl autocompletion to the bash-completion:

```
kubectl completion bash > $(brew --prefix)/etc/bash_completion.d/kubectl
```

The Homebrew project is independent from Kubernetes, so the bash-completion packages are not guaranteed to work.

Using Oh-My-Zsh

When using Oh-My-Zsh, edit the ~/.zshrc file and update the `plugins=` line to include the kubectl plugin.

```
plugins=(git zsh-completions kubectl)
```

What’s next

Learn how to launch and expose your application.

Install Minikube

This page shows how to use install Minikube.

Before you begin

VT-x or AMD-v virtualization must be enabled in your computer's BIOS.

Install a Hypervisor

If you do not already have a hypervisor installed, install one now.

- For OS X, install xhyve driver, VirtualBox, or VMware Fusion.
- For Linux, install VirtualBox or KVM.
- For Windows, install VirtualBox or Hyper-V.

Install kubectl

- Install kubectl.

Install Minikube

- Install Minikube according to the instructions for the latest release.

What’s next

- Running Kubernetes Locally via Minikube

Installing kubeadm

This page shows how to use install kubeadm.

Before you begin

- One or more machines running Ubuntu 16.04+, Debian 9, CentOS 7, RHEL 7, Fedora 25/26 (best-effort) or HypriotOS v1.0.1+
- 1GB or more of RAM per machine (any less will leave little room for your apps)
- Full network connectivity between all machines in the cluster (public or private network is fine)
- Unique MAC address and product_uuid for every node
- Certain ports are open on your machines. See the section below for more details
- Swap disabled. You must disable swap in order for the kubelet to work properly.

Check required ports

Master node(s)

Port Range	Purpose
6443*	Kubernetes API server
2379-2380	etcd server client API

Port Range	Purpose
10250	Kubelet API
10251	kube-scheduler
10252	kube-controller-manager
10255	Read-only Kubelet API (Heapster)

Worker node(s)

Port Range	Purpose
10250	Kubelet API
10255	Read-only Kubelet API (Heapster)
30000-32767	Default port range for NodePort Services. Typically, these ports would need to be exposed to external load-balancers, or other external consumers of the application itself.

Any port numbers marked with * are overridable, so you will need to ensure any custom ports you provide are also open.

Although etcd ports are included in master nodes, you can also host your own etcd cluster externally on custom ports.

The pod network plugin you use (see below) may also require certain ports to be open. Since this differs with each pod network plugin, please see the documentation for the plugins about what port(s) those need.

Installing Docker

On each of your machines, install Docker. Version v1.12 is recommended, but v1.11, v1.13 and 17.03 are known to work as well. Versions 17.06+ *might work*, but have not yet been tested and verified by the Kubernetes node team.

You can use the following commands to install Docker on your system:

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

Install Docker from Ubuntu's repositories:

```
apt-get update
apt-get install -y docker.io
```

or install Docker CE 17.03 from Docker's repositories for Ubuntu or Debian:

```
apt-get update && apt-get install -y curl apt-transport-https
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/docker.list
deb https://download.docker.com/linux/$(lsb_release -si | tr '[:upper:]' '[:
EOF
apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce
```

Installing kubeadm, kubelet and kubectl

You will install these packages on all of your machines:

- **kubeadm** : the command to bootstrap the cluster.
- **kubelet** : the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- **kubectl** : the command line util to talk to your cluster.

Please proceed with executing the following commands based on your OS as **root** . You may become the **root** user by executing **sudo -i** after SSH-ing to each host.

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
```

The kubelet is now restarting every few seconds, as it waits in a crashloop for kubeadm to tell it what to do.

What's next

- Using kubeadm to Create a Cluster

Assign Memory Resources to Containers and Pods

This page shows how to assign a memory *request* and a memory *limit* to a Container. A Container is guaranteed to have as much memory as it requests, but is not allowed to use more memory than its limit.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 300 MiB of memory.

A few of the steps on this page require that the Heapster service is running in your cluster. But if you don't have Heapster running, you can do most of the steps, and it won't be a problem if you skip the Heapster steps.

To see whether the Heapster service is running, enter this command:

```
kubectl get services --namespace=kube-system
```

If the Heapster service is running, it shows in the output:

```
NAMESPACE    NAME      CLUSTER-IP    EXTERNAL-IP  PORT(S)  AGE
kube-system  heapster  10.11.240.9   <none>       80/TCP   6d
```

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace mem-example
```

Specify a memory request and a memory limit

To specify a memory request for a Container, include the `resources:requests` field in the Container's resource manifest. To specify a memory limit, include `resources:limits`.

In this exercise, you create a Pod that has one Container. The Container has a memory request of 100 MiB and a memory limit of 200 MiB. Here's the configuration file for the Pod:

memory-request-limit.yaml

memory-request-limit.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
spec:
  containers:
  - name: memory-demo-ctr
    image: vish/stress
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
    args:
    - -mem-total
    - 150Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

In the configuration file, the `args` section provides arguments for the Container when it starts. The `-mem-total 150Mi` argument tells the Container to attempt to allocate 150 MiB of memory.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request
```

Verify that the Pod's Container is running:

```
kubectl get pod memory-demo --namespace=mem-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo --output=yaml --namespace=mem-example
```

The output shows that the one Container in the Pod has a memory request of 100 MiB and a memory limit of 200 MiB.

```
...
resources:
  limits:
    memory: 200Mi
  requests:
    memory: 100Mi
...
```

Start a proxy so that you can call the Heapster service:

```
kubectl proxy
```

In another command window, get the memory usage from the Heapster service:

```
curl http://localhost:8001/api/v1/proxy/namespaces/kube-system/services/heapster/a
```

The output shows that the Pod is using about 162,900,000 bytes of memory, which is about 150 MiB. This is greater than the Pod's 100 MiB request, but within the Pod's 200 MiB limit.

```
{
 "timestamp": "2017-06-20T18:54:00Z",
 "value": 162856960
}
```

Delete your Pod:

```
kubectl delete pod memory-demo --namespace=mem-example
```

Exceed a Container's memory limit

A Container can exceed its memory request if the Node has memory available. But a Container is not allowed to use more than its memory limit. If a Container allocates more memory than its limit, the

Container becomes a candidate for termination. If the Container continues to consume memory beyond its limit, the Container is terminated. If a terminated Container is restartable, the kubelet will restart it, as with any other type of runtime failure.

In this exercise, you create a Pod that attempts to allocate more memory than its limit. Here is the configuration file for a Pod that has one Container. The Container has a memory request of 50 MiB and a memory limit of 100 MiB.

memory-request-limit-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
spec:
  containers:
  - name: memory-demo-2-ctr
    image: vish/stress
    resources:
      requests:
        memory: 50Mi
      limits:
        memory: "100Mi"
    args:
    - -mem-total
    - 250Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

In the configuration file, in the `args` section, you can see that the Container will attempt to allocate 250 MiB of memory, which is well above the 100 MiB limit.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request
```

View detailed information about the Pod:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

At this point, the Container might be running, or it might have been killed. If the Container has not yet been killed, repeat the preceding command until you see that the Container has been killed:

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

Get a more detailed view of the Container's status:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

The output shows that the Container has been killed because it is out of memory (OOM).

```
lastState:
   terminated:
     containerID: docker://65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd229
     exitCode: 137
     finishedAt: 2017-06-20T20:52:19Z
     reason: OOMKilled
     startedAt: null
```

The Container in this exercise is restartable, so the kubelet will restart it. Enter this command several times to see that the Container gets repeatedly killed and restarted:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container gets killed, restarted, killed again, restarted again, and so on:

```
stevepe@sperry-1:~/steveperry-53.github.io$ kubectl get pod memory-demo-2 --namesp
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          37s
stevepe@sperry-1:~/steveperry-53.github.io$ kubectl get pod memory-demo-2 --namesp
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-2   1/1       Running   2          40s
```

View detailed information about the Pod's history:

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container starts and fails repeatedly:

```
... Normal  Created   Created container with id 66a3a20aa7980e61be4922780bf9d24d1a
... Warning BackOff   Back-off restarting failed container
```

View detailed information about your cluster's Nodes:

```
kubectl describe nodes
```

The output includes a record of the Container being killed because of an out-of-memory condition:

```
Warning OOMKilling  Memory cgroup out of memory: Kill process 4481 (stress) score
```

Delete your Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

Specify a memory request that is too big for your Nodes

Memory requests and limits are associated with Containers, but it is useful to think of a Pod as having a memory request and limit. The memory request for the Pod is the sum of the memory requests for all the Containers in the Pod. Likewise, the memory limit for the Pod is the sum of the limits of all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough available memory to satisfy the Pod's memory request.

In this exercise, you create a Pod that has a memory request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container. The

Container requests 1000 GiB of memory, which is likely to exceed the capacity of any Node in your cluster.

memory-request-limit-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
spec:
  containers:
  - name: memory-demo-3-ctr
    image: vish/stress
    resources:
      limits:
        memory: "1000Gi"
      requests:
        memory: "1000Gi"
    args:
    - -mem-total
    - 150Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request
```

View the Pod’s status:

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

The output shows that the Pod’s status is PENDING. That is, the Pod has not been scheduled to run on any Node, and it will remain in the PENDING state indefinitely:

```
kubectl get pod memory-demo-3 --namespace=mem-example
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-3   0/1       Pending   0          25s
```

View detailed information about the Pod, including events:

```
kubectl describe pod memory-demo-3 --namespace=mem-example
```

The output shows that the Container cannot be scheduled because of insufficient memory on the Nodes:

```
Events:
  ...  Reason            Message
       ------            -------
  ...  FailedScheduling  No nodes are available that match all of the following pr
```

Memory units

The memory resource is measured in bytes. You can express memory as a plain integer or a fixed-point integer with one of these suffixes: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent approximately the same value:

```
128974848, 129e6, 129M , 123Mi
```

Delete your Pod:

```
kubectl delete pod memory-demo-3 --namespace=mem-example
```

If you don’t specify a memory limit

If you don’t specify a memory limit for a Container, then one of these situations applies:

- The Container has no upper bound on the amount of memory it uses. The Container could use all of the memory available on the Node where it is running.

- The Container is running in a namespace that has a default memory limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the memory limit.

Motivation for memory requests and limits

By configuring memory requests and limits for the Containers that run in your cluster, you can make efficient use of the memory resources available on your cluster's Nodes. By keeping a Pod's memory request low, you give the Pod a good chance of being scheduled. By having a memory limit that is greater than the memory request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of memory that happens to be available.
- The amount of memory a Pod can use during a burst is limited to some reasonable amount.

Clean up

Delete your namespace. This deletes all the Pods that you created for this task:

```
kubectl delete namespace mem-example
```

What's next

For app developers

- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace

- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

Assign CPU Resources to Containers and Pods

This page shows how to assign a CPU *request* and a CPU *limit* to a Container. A Container is guaranteed to have as much CPU as it requests, but is not allowed to use more CPU than its limit.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 1 cpu.

A few of the steps on this page require that the Heapster service is running in your cluster. But if you don't have Heapster running, you can do most of the steps, and it won't be a problem if you skip the Heapster steps.

To see whether the Heapster service is running, enter this command:

```
kubectl get services --namespace=kube-system
```

If the heapster service is running, it shows in the output:

```
NAMESPACE    NAME      CLUSTER-IP    EXTERNAL-IP  PORT(S)  AGE
kube-system  heapster  10.11.240.9   <none>       80/TCP   6d
```

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace cpu-example
```

Specify a CPU request and a CPU limit

To specify a CPU request for a Container, include the `resources:requests` field in the Container's resource manifest. To specify a CPU limit, include `resources:limits`.

In this exercise, you create a Pod that has one Container. The Container has a CPU request of 0.5 cpu and a CPU limit of 1 cpu. Here's the configuration file for the Pod:

cpu-request-limit.yaml

cpu-request-limit.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
spec:
  containers:
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
    args:
    - -cpus
    - "2"
```

In the configuration file, the `args` section provides arguments for the Container when it starts. The `-cpus "2"` argument tells the Container to attempt to use 2 cpus.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-li
```

Verify that the Pod's Container is running:

```
kubectl get pod cpu-demo --namespace=cpu-example
```

View detailed information about the Pod:

```
kubectl get pod cpu-demo --output=yaml --namespace=cpu-example
```

The output shows that the one Container in the Pod has a CPU request of 500 millicpu and a CPU limit of 1 cpu.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 500m
```

Start a proxy so that you can call the heapster service:

```
kubectl proxy
```

In another command window, get the CPU usage rate from the heapster service:

```
curl http://localhost:8001/api/v1/proxy/namespaces/kube-system/services/heapster/a
```

The output shows that the Pod is using 974 millicpu, which is just a bit less than the limit of 1 cpu specified in the Pod's configuration file.

```
{
 "timestamp": "2017-06-22T18:48:00Z",
 "value": 974
}
```

Recall that by setting `-cpu "2"`, you configured the Container to attempt to use 2 cpus. But the Container is only being allowed to use about 1 cpu. The Container's CPU use is being throttled, because the Container is attempting to use more CPU resources than its limit.

> **Note:** There's another possible explanation for the CPU throttling. The Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require that each of your Nodes has at least 1 cpu. If your Container is running on a Node that has only 1 cpu, the Container cannot use more than 1 cpu regardless of the CPU limit specified for the Container.

CPU units

The CPU resource is measured in *cpu* units. One cpu, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

Fractional values are allowed. A Container that requests 0.5 cpu is guaranteed half as much CPU as a Container that requests 1 cpu. You can use the suffix m to mean milli. For example 100m cpu, 100 millicpu, and 0.1 cpu are all the same. Precision finer than 1m is not allowed.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

Delete your Pod:

```
kubectl delete pod cpu-demo --namespace=cpu-example
```

Specify a CPU request that is too big for your Nodes

CPU requests and limits are associated with Containers, but it is useful to think of a Pod as having a CPU request and limit. The CPU request for a Pod is the sum of the CPU requests for all the Containers in the Pod. Likewise, the CPU limit for a Pod is the sum of the CPU limits for all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough CPU resources available to satisfy the Pod's CPU request.

In this exercise, you create a Pod that has a CPU request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container. The Container requests 100 cpu, which is likely to exceed the capacity of any Node in your cluster.

cpu-request-limit-2.yaml

cpu-request-limit-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
spec:
  containers:
  - name: cpu-demo-ctr-2
    image: vish/stress
    resources:
      limits:
        cpu: "100"
      requests:
        cpu: "100"
    args:
    - -cpus
    - "2"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-li
```

View the Pod's status:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Pod's status is Pending. That is, the Pod has not been scheduled to run on any Node, and it will remain in the Pending state indefinitely:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
NAME         READY     STATUS    RESTARTS   AGE
cpu-demo-2   0/1       Pending   0          7m
```

View detailed information about the Pod, including events:

```
kubectl describe pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Container cannot be scheduled because of insufficient CPU resources on the Nodes:

```
Events:
  Reason                        Message
  ------                        -------
  FailedScheduling      No nodes are available that match all of the following pre
```

Delete your Pod:

```
kubectl delete pod cpu-demo-2 --namespace=cpu-example
```

If you don't specify a CPU limit

If you don't specify a CPU limit for a Container, then one of these situations applies:

- The Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available on the Node where it is running.
- The Container is running in a namespace that has a default CPU limit, and the Container is automatically assigned the default limit. Cluster administrators can use a LimitRange to specify a default value for the CPU limit.

Motivation for CPU requests and limits

By configuring the CPU requests and limits of the Containers that run in your cluster, you can make efficient use of the CPU resources available on your cluster's Nodes. By keeping a Pod's CPU request low, you give the Pod a good chance of being scheduled. By having a CPU limit that is greater than the CPU request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of CPU resources that happen to be available.
- The amount of CPU resources a Pod can use during a burst is limited to some reasonable amount.

Clean up

Delete your namespace:

```
kubectl delete namespace cpu-example
```

What's next

For app developers

- Assign Memory Resources to Containers and Pods
- Configure Quality of Service for Pods

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

Configure Quality of Service for Pods

This page shows how to configure Pods so that they will be assigned particular Quality of Service (QoS) classes. Kubernetes uses QoS classes to make decisions about scheduling and evicting Pods.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

QoS classes

When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

- Guaranteed
- Burstable

- BestEffort

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace qos-example
```

Create a Pod that gets assigned a QoS class of Guaranteed

For a Pod to be given a QoS class of Guaranteed:

- Every Container in the Pod must have a memory limit and a memory request, and they must be the same.
- Every Container in the Pod must have a cpu limit and a cpu request, and they must be the same.

Here is the configuration file for a Pod that has one Container. The Container has a memory limit and a memory request, both equal to 200 MiB. The Container has a cpu limit and a cpu request, both equal to 700 millicpu:

qos-pod.yaml

qos-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod.yaml -
```

View detailed information about the Pod:

```
kubectl get pod qos-demo --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of Guaranteed. The output also verifies that the Pod's Container has a memory request that matches its memory limit, and it has a cpu request that matches its cpu limit.

```
spec:
  containers:
    ...
    resources:
      limits:
        cpu: 700m
        memory: 200Mi
      requests:
        cpu: 700m
        memory: 200Mi
...
  qosClass: Guaranteed
```

Note: If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own cpu limit, but does not specify a cpu request, Kubernetes automatically assigns a cpu request that matches the limit.

Delete your Pod:

```
kubectl delete pod qos-demo --namespace=qos-example
```

Create a Pod that gets assigned a QoS class of Burstable

A Pod is given a QoS class of Burstable if:

- The Pod does not meet the criteria for QoS class Guaranteed.
- At least one Container in the Pod has a memory or cpu request.

Here is the configuration file for a Pod that has one Container. The Container has a memory limit of 200 MiB and a memory request of 100 MiB.

qos-pod-2.yaml

qos-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-2
spec:
  containers:
  - name: qos-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-2.yaml
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-2 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of Burstable.

```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: qos-demo-2-ctr
    resources:
      limits:
        memory: 200Mi
      requests:
        memory: 100Mi
...
  qosClass: Burstable
```

Delete your Pod:

```
kubectl delete pod qos-demo-2 --namespace=qos-example
```

Create a Pod that gets assigned a QoS class of BestEffort

For a Pod to be given a QoS class of BestEffort, the Containers in the Pod must not have any memory or cpu limits or requests.

Here is the configuration file for a Pod that has one Container. The Container has no memory or cpu limits or requests:

qos-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-3
spec:
  containers:
  - name: qos-demo-3-ctr
    image: nginx
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-3.yaml
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-3 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of BestEffort.

```
spec:
  containers:
    ...
    resources: {}
  ...
  qosClass: BestEffort
```

Delete your Pod:

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

Create a Pod that has two Containers

Here is the configuration file for a Pod that has two Containers. One container specifies a memory request of 200 MiB. The other Container does not specify any requests or limits.

qos-pod-4.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-4
spec:
  containers:

  - name: qos-demo-4-ctr-1
    image: nginx
    resources:
      requests:
        memory: "200Mi"

  - name: qos-demo-4-ctr-2
    image: redis
```

Notice that this Pod meets the criteria for QoS class Burstable. That is, it does not meet the criteria for QoS class Guaranteed, and one of its Containers has a memory request.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-4.yaml
```

View detailed information about the Pod:

```
kubectl get pod qos-demo-4 --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of Burstable:

```
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
    ...
  qosClass: Burstable
```

Delete your Pod:

```
kubectl delete pod qos-demo-4 --namespace=qos-example
```

Clean up

Delete your namespace:

```
kubectl delete namespace qos-example
```

What’s next

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

Assign Opaque Integer Resources to a Container

This page shows how to assign opaque integer resources to a Container.

DEPRECATION NOTICE: As of `Kubernetes v1.8`, this has been deprecated

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Before you do this exercise, do the exercise in Advertise Opaque Integer Resources for a Node. That will configure one of your Nodes to advertise a dongle resource.

Assign an opaque integer resource to a Pod

To request an opaque integer resource, include the `resources:requests` field in your Container manifest. Opaque integer resources have the prefix `pod.alpha.kubernetes.io/opaque-int-resource-`.

Here is the configuration file for a Pod that has one Container:

oir-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: oir-demo
spec:
  containers:
  - name: oir-demo-ctr
    image: nginx
    resources:
      requests:
        pod.alpha.kubernetes.io/opaque-int-resource-dongle: 3
```

In the configuration file, you can see that the Container requests 3 dongles.

Create a Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/oir-pod.yaml
```

Verify that the Pod is running:

```
kubectl get pod oir-demo
```

Describe the Pod:

```
kubectl describe pod oir-demo
```

The output shows dongle requests:

```
Requests:
  pod.alpha.kubernetes.io/opaque-int-resource-dongle: 3
```

Attempt to create a second Pod

Here is the configuration file for a Pod that has one Container. The Container requests two dongles.

oir-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: oir-demo-2
spec:
  containers:
  - name: oir-demo-2-ctr
    image: nginx
    resources:
      requests:
        pod.alpha.kubernetes.io/opaque-int-resource-dongle: 2
```

Kubernetes will not be able to satisfy the request for two dongles, because the first Pod used three of the four available dongles.

Attempt to create a Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/oir-pod-2.yaml
```

Describe the Pod

```
kubectl describe pod oir-demo-2
```

The output shows that the Pod cannot be scheduled, because there is no Node that has 2 dongles available:

```
Conditions:
  Type    Status
  PodScheduled  False
...
Events:
  ...
  ... Warning   FailedScheduling  pod (oir-demo-2) failed to fit in any node
fit failure summary on nodes : Insufficient pod.alpha.kubernetes.io/opaque-int-res
```

View the Pod status:

```
kubectl get pod oir-demo-2
```

The output shows that the Pod was created, but not scheduled to run on a Node. It has a status of Pending:

```
NAME          READY     STATUS    RESTARTS   AGE
oir-demo-2    0/1       Pending   0          6m
```

Clean up

Delete the Pod that you created for this exercise:

```
kubectl delete pod oir-demo
```

What’s next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Advertise Opaque Integer Resources for a Node

Configure a Pod to Use a Volume for Storage

This page shows how to configure a Pod to use a Volume for storage.

A Container's file system lives only as long as the Container does, so when a Container terminates and restarts, changes to the filesystem are lost. For more consistent storage that is independent of the Container, you can use a Volume. This is especially important for stateful applications, such as key-value stores and databases. For example, Redis is a key-value cache and store.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Configure a volume for a Pod

In this exercise, you create a Pod that runs one Container. This Pod has a Volume of type emptyDir that lasts for the life of the Pod, even if the Container terminates and restarts. Here is the configuration file for the Pod:

pod-redis.yaml

pod-redis.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

1. Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/pod-redis.
```

2. Verify that the Pod's Container is running, and then watch for changes to the Pod:

```
kubectl get pod redis --watch
```

The output looks like this:

```
NAME      READY     STATUS    RESTARTS   AGE
redis     1/1       Running   0          13s
```

3. In another terminal, get a shell to the running Container:

```
kubectl exec -it redis -- /bin/bash
```

4. In your shell, go to `/data/redis`, and create a file:

```
root@redis:/data/redis# echo Hello > test-file
```

5. In your shell, list the running processes:

   ```
   root@redis:/data/redis# ps aux
   ```

   The output is similar to this:

   ```
   USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
   redis        1  0.1  0.1  33308  3828 ?        Ssl  00:46   0:00 redis-server
   root        12  0.0  0.0  20228  3020 ?        Ss   00:47   0:00 /bin/bash
   root        15  0.0  0.0  17500  2072 ?        R+   00:48   0:00 ps aux
   ```

6. In your shell, kill the redis process:

   ```
   root@redis:/data/redis# kill <pid>
   ```

   where `<pid>` is the redis process ID (PID).

7. In your original terminal, watch for changes to the redis Pod. Eventually, you will see something like this:

   ```
   NAME      READY     STATUS     RESTARTS   AGE
   redis     1/1       Running    0          13s
   redis     0/1       Completed  0         6m
   redis     1/1       Running    1         6m
   ```

At this point, the Container has terminated and restarted. This is because the redis Pod has a restartPolicy of `Always`.

1. Get a shell into the restarted Container:

```
kubectl exec -it redis -- /bin/bash
```

2. In your shell, goto `/data/redis`, and verify that `test-file` is still there.

What's next

- See Volume.
- See Pod.
- In addition to the local disk storage provided by `emptyDir`, Kubernetes supports many different network-attached storage solutions, including PD on GCE and EBS on EC2, which are preferred for critical data, and will handle details such as mounting and unmounting the devices on the nodes. See Volumes for more details.

Configure a Pod to Use a PersistentVolume for Storage

This page shows how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. A cluster administrator creates a PersistentVolume that is backed by physical storage. The administrator does not associate the volume with any Pod.
2. A cluster user creates a PersistentVolumeClaim, which gets automatically bound to a suitable PersistentVolume.
3. The user creates a Pod that uses the PersistentVolumeClaim as storage.

Before you begin

- You need to have a Kubernetes cluster that has only one Node, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a single-node cluster, you can create one by using Minikube.
- Familiarize yourself with the material in Persistent Volumes.

Create an index.html file on your Node

Open a shell to the Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

In your shell, create a `/tmp/data` directory:

```
mkdir /tmp/data
```

In the `/tmp/data` directory, create an `index.html` file:

```
echo 'Hello from Kubernetes storage' > /tmp/data/index.html
```

Create a PersistentVolume

In this exercise, you create a *hostPath* PersistentVolume. Kubernetes supports hostPath for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, you would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use StorageClasses to set up dynamic provisioning.

Here is the configuration file for the hostPath PersistentVolume:

task-pv-volume.yaml

task-pv-volume.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

The configuration file specifies that the volume is at `/tmp/data` on the the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of `ReadWriteOnce`, which means the volume can be mounted as read-write by a single Node. It defines the StorageClass name `manual` for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Create the PersistentVolume:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/task-pv-volume
```

View information about the PersistentVolume:

```
kubectl get pv task-pv-volume
```

The output shows that the PersistentVolume has a `STATUS` of `Available`. This means it has not yet been bound to a PersistentVolumeClaim.

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS      CLAIM     ST
task-pv-volume   10Gi       RWO           Retain          Available             ma
```

Create a PersistentVolumeClaim

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this exercise, you create a PersistentVolumeClaim that requests a volume of at least three gibibytes that can provide read-write access for at least one Node.

Here is the configuration file for the PersistentVolumeClaim:

task-pv-claim.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/task-pv-claim.
```

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

```
kubectl get pv task-pv-volume
```

Now the output shows a `STATUS` of `Bound`.

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    CLAIM
task-pv-volume   10Gi       RWO           Retain          Bound     default/task-p
```

Look at the PersistentVolumeClaim:

```
kubectl get pvc task-pv-claim
```

The output shows that the PersistentVolumeClaim is bound to your PersistentVolume, `task-pv-volume`.

```
NAME            STATUS    VOLUME           CAPACITY   ACCESSMODES   STORAGECLASS
task-pv-claim   Bound     task-pv-volume   10Gi       RWO           manual
```

Create a Pod

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

task-pv-pod.yaml

task-pv-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:

  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: task-pv-claim

  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: task-pv-storage
```

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/task-pv-pod.ya
```

Verify that the Container in the Pod is running;

```
kubectl get pod task-pv-pod
```

Get a shell to the Container running in your Pod:

```
kubectl exec -it task-pv-pod -- /bin/bash
```

In your shell, verify that nginx is serving the `index.html` file from the hostPath volume:

```
root@task-pv-pod:/# apt-get update
root@task-pv-pod:/# apt-get install curl
root@task-pv-pod:/# curl localhost
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

Access control

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can annotate a PersistentVolume with a GID. Then the GID is automatically added to any Pod that uses the PersistentVolume.

Use the `pv.beta.kubernetes.io/gid` annotation as follows:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

When a Pod consumes a PersistentVolume that has a GID annotation, the annotated GID is applied to all Containers in the Pod in the same way that GIDs specified in the Pod's security context are. Every GID, whether it originates from a PersistentVolume annotation or the Pod's specification, is applied to the first process run in each Container.

Note: When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself.

What's next

- Learn more about PersistentVolumes.
- Read the Persistent Storage design document.

Reference

- PersistentVolume
- PersistentVolumeSpec
- PersistentVolumeClaim
- PersistentVolumeClaimSpec

Configure a Pod to Use a Projected Volume for Storage

This page shows how to use a `projected` volume to mount several existing volume sources into the same directory. Currently, `secret`, `configMap`, and `downwardAPI` volumes can be projected.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Configure a projected volume for a pod

In this exercise, you create username and password Secrets from local files. You then create a Pod that runs one Container, using a `projected` Volume to mount the Secrets into the same shared directory.

Here is the configuration file for the Pod:

projected-volume.yaml

projected-volume.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: user
      - secret:
          name: pass
```

1. Create the Secrets:

```
# Create files containing the username and password:
echo -n "admin" > ./username.txt
echo -n "1f2d1e2e67df" > ./password.txt

# Package these files into secrets:
kubectl create secret generic user --from-file=./username.txt
kubectl create secret generic pass --from-file=./password.txt
```

2. Create the Pod:

```
kubectl create -f projected-volume.yaml
```

3. Verify that the Pod's Container is running, and then watch for changes to the Pod:

   ```
   kubectl get --watch pod test-projected-volume
   ```

   The output looks like this:

   ```
   NAME                    READY     STATUS    RESTARTS   AGE
   test-projected-volume   1/1       Running   0          14s
   ```

4. In another terminal, get a shell to the running Container:

   ```
   kubectl exec -it test-projected-volume -- /bin/sh
   ```

5. In your shell, verify that the `projected-volume` directory contains your projected sources:

   ```
   / # ls /projected-volume/
   ```

What's next

- Learn more about `projected` volumes.
- Read the the all-in-one volume design document.

Configure a Security Context for a Pod or Container

A security context defines privilege and access control settings for a Pod or Container. Security context settings include:

- Discretionary Access Control: Permission to access an object, like a file, is based on user ID (UID) and group ID (GID).
- Security Enhanced Linux (SELinux): Objects are assigned security labels.
- Running as privileged or unprivileged.
- Linux Capabilities: Give a process some privileges, but not all the privileges of the root user.
- AppArmor: Use program profiles to restrict the capabilities of individual programs.
- Seccomp: Limit a process's access to open file descriptors.
- AllowPrivilegeEscalation: Controls whether a process can gain more privileges than its parent process. This bool directly controls whether the `no_new_privs` flag gets set on the container process. AllowPrivilegeEscalation is true always when the container is: 1) run as Privileged OR 2) has `CAP_SYS_ADMIN`.

For more information about security mechanisms in Linux, see Overview of Linux Kernel Security Features

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Set the security context for a Pod

To specify security settings for a Pod, include the `securityContext` field in the Pod specification. The `securityContext` field is a PodSecurityContext object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a `securityContext` and an `emptyDir` volume:

security-context.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: gcr.io/google-samples/node-hello:1.0
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
      allowPrivilegeEscalation: false
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, the first process runs with user ID 1000. The `fsGroup` field specifies that group ID 2000 is associated with all Containers in the Pod. Group ID 2000 is also associated with the volume mounted at `/data/demo` and with any files created in that volume.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/security-conte
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

```
USER   PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
1000     1  0.0  0.0   4336   724 ?     Ss   18:16   0:00 /bin/sh -c node server.j
1000     5  0.2  0.6 772124 22768 ?     Sl   18:16   0:00 node server.js
...
```

In your shell, navigate to `/data`, and list the one directory:

```
cd /data
ls -l
```

The output shows that the `/data/demo` directory has group ID 2000, which is the value of `fsGroup`.

```
drwxrwsrwx 2 root 2000 4096 Jun  6 20:08 demo
```

In your shell, navigate to `/data/demo`, and create a file:

```
cd demo
echo hello > testfile
```

List the file in the `/data/demo` directory:

```
ls -l
```

The output shows that `testfile` has group ID 2000, which is the value of `fsGroup`.

```
-rw-r--r-- 1 1000 2000 6 Jun  6 20:08 testfile
```

Exit your shell:

```
exit
```

Set the security context for a Container

To specify security settings for a Container, include the `securityContext` field in the Container manifest. The `securityContext` field is a SecurityContext object. Security settings that you specify for a Container apply only to the individual Container, and they override settings made at the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a `securityContext` field:

security-context-2.yaml

security-context-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/security-conte
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 2000. This is the value of `runAsUser` specified for the Container. It overrides the value 1000 that is specified for the Pod.

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
2000         1  0.0  0.0   4336   764 ?        Ss   20:36   0:00 /bin/sh -c node s
2000         8  0.1  0.5 772124 22604 ?        Sl   20:36   0:00 node server.js
...
```

Exit your shell:

```
exit
```

Set capabilities for a Container

With Linux capabilities, you can grant certain privileges to a process without granting all the privileges of the root user. To add or remove Linux capabilities for a Container, include the `capabilities` field in the `securityContext` section of the Container manifest.

First, see what happens when you don't include a `capabilities` field. Here is configuration file that does not add or remove any Container capabilities:

security-context-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
  - name: sec-ctx-3
    image: gcr.io/google-samples/node-hello:1.0
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/security-conte
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-3
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-3 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows the process IDs (PIDs) for the Container:

```
USER   PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root     1  0.0  0.0   4336   796 ?     Ss   18:17   0:00 /bin/sh -c node server.js
root     5  0.1  0.5 772124 22700 ?     Sl   18:17   0:00 node server.js
```

In your shell, view the status for process 1:

```
cd /proc/1
cat status
```

The output shows the capabilities bitmap for the process:

```
...
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
...
```

Make a note of the capabilities bitmap, and then exit your shell:

```
exit
```

Next, run a Container that is the same as the preceding container, except that it has additional capabilities set.

Here is the configuration file for a Pod that runs one Container. The configuration adds the `CAP_NET_ADMIN` and `CAP_SYS_TIME` capabilities:

security-context-4.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/security-conte
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-4 -- sh
```

In your shell, view the capabilities for process 1:

```
cd /proc/1
cat status
```

The output shows capabilities bitmap for the process:

```
...
CapPrm: 00000000aa0435fb
CapEff: 00000000aa0435fb
...
```

Compare the capabilities of the two Containers:

```
00000000a80425fb
00000000aa0435fb
```

In the capability bitmap of the first container, bits 12 and 25 are clear. In the second container, bits 12 and 25 are set. Bit 12 is `CAP_NET_ADMIN`, and bit 25 is `CAP_SYS_TIME`. See capability.h for definitions of the capability constants.

> **Note:** Linux capability constants have the form `CAP_XXX`. But when you list capabilities in your Container manifest, you must omit the `CAP_` portion of the constant. For example, to add `CAP_SYS_TIME`, include `SYS_TIME` in your list of capabilities.

Assign SELinux labels to a Container

To assign SELinux labels to a Container, include the `seLinuxOptions` field in the `securityContext` section of your Pod or Container manifest. The `seLinuxOptions` field is an SELinuxOptions object. Here's an example that applies an SELinux level:

```
...
securityContext:
  seLinuxOptions:
    level: "s0:c123,c456"
```

> **Note:** To assign SELinux labels, the SELinux security module must be loaded on the host operating system.

Discussion

The security context for a Pod applies to the Pod's Containers and also to the Pod's Volumes when applicable. Specifically `fsGroup` and `seLinuxOptions` are applied to Volumes as follows:

- `fsGroup` : Volumes that support ownership management are modified to be owned and writable by the GID specified in `fsGroup` . See the Ownership Management design document for more details.
- `seLinuxOptions` : Volumes that support SELinux labeling are relabeled to be accessible by the label specified under `seLinuxOptions` . Usually you only need to set the `level` section. This sets the Multi-Category Security (MCS) label given to all Containers in the Pod as well as the Volumes.

Warning: After you specify an MCS label for a Pod, all Pods with the same label can access the Volume. If you need inter-Pod protection, you must assign a unique MCS label to each Pod.

What's next

- PodSecurityContext
- SecurityContext
- Tuning Docker with the newest security enhancements
- Security Contexts design document
- Ownership Management design document
- Pod Security Policies
- AllowPrivilegeEscalation design document

Configure Service Accounts for Pods

A service account provides an identity for processes that run in a Pod.

This is a user introduction to Service Accounts. See also the Cluster Admin Guide to Service Accounts.

Note: This document describes how service accounts behave in a cluster set up as recommended by the Kubernetes project. Your cluster administrator may have customized the behavior in your cluster, in which case this documentation may not apply.

When you (a human) access the cluster (e.g. using `kubectl`), you are authenticated by the apiserver as a particular User Account (currently this is usually `admin`, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (e.g. `default`).

Use the Default Service Account to access the API server.

When you create a pod, if you do not specify a service account, it is automatically assigned the `default` service account in the same namespace. If you get the raw json or yaml for a pod you have created (e.g. `kubectl get pods/podname -o yaml`), you can see the `spec.serviceAccountName` field has been automatically set.

You can access the API from inside a pod using automatically mounted service account credentials, as described in Accessing the Cluster. The API permissions a service account has depend on the authorization plugin and policy in use.

In version 1.6+, you can opt out of automounting API credentials for a service account by setting `automountServiceAccountToken: false` on the service account:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

In version 1.6+, you can also opt out of automounting API credentials for a particular pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
  ...
```

The pod spec takes precedence over the service account if both specify a `automountServiceAccountToken` value.

Use Multiple Service Accounts.

Every namespace has a default service account resource called `default`. You can list this and any other serviceAccount resources in the namespace with this command:

```
$ kubectl get serviceAccounts
NAME      SECRETS    AGE
default   1          1d
```

You can create additional ServiceAccount objects like this:

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

If you get a complete dump of the service account object, like this:

```
$ kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  selfLink: /api/v1/namespaces/default/serviceaccounts/build-robot
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

then you will see that a token has automatically been created and is referenced by the service account.

You may use authorization plugins to set permissions on service accounts.

To use a non-default service account, simply set the `spec.serviceAccountName` field of a pod to the name of the service account you wish to use.

The service account has to exist at the time the pod is created, or it will be rejected.

You cannot update the service account of an already created pod.

You can clean up the service account from this example like this:

```
$ kubectl delete serviceaccount/build-robot
```

Manually create a service account API token.

Suppose we have an existing service account named “build-robot” as mentioned above, and we create a new secret manually.

```
$ cat > /tmp/build-robot-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

Now you can confirm that the newly built secret is populated with an API token for the “build-robot” service account.

Any tokens for non-existent service accounts will be cleaned up by the token controller.

```
$ kubectl describe secrets/build-robot-secret
Name:           build-robot-secret
Namespace:      default
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=build-robot
                kubernetes.io/service-account.uid=da68f9c6-9d26-11e7-b84e-002dc528

Type:   kubernetes.io/service-account-token

Data
====
ca.crt:         1338 bytes
namespace:      7 bytes
token:          ...
```

Note: The content of `token` is elided here.

Add ImagePullSecrets to a service account

First, create an imagePullSecret, as described here. Next, verify it has been created. For example:

```
$ kubectl get secrets myregistrykey
NAME             TYPE                              DATA    AGE
myregistrykey    kubernetes.io/.dockerconfigjson   1       1d
```

Next, modify the default service account for the namespace to use this secret as an imagePullSecret.

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistr
```

Interactive version requiring manual edit:

```
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
$ vi sa.yaml
[editor session not shown]
[delete line with key "resourceVersion"]
[add lines with "imagePullSecret:"]
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
$ kubectl replace serviceaccount default -f ./sa.yaml
serviceaccounts/default
```

Now, any new pods created in the current namespace will have this added to their spec:

```
spec:
  imagePullSecrets:
  - name: myregistrykey
```

Pull an Image from a Private Registry

This page shows how to create a Pod that uses a Secret to pull an image from a private Docker registry or repository.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- To do this exercise, you need a Docker ID and password.

Log in to Docker

```
docker login
```

When prompted, enter your Docker username and password.

The login process creates or updates a `config.json` file that holds an authorization token.

View the `config.json` file:

```
cat ~/.docker/config.json
```

The output contains a section similar to this:

```
{
    "auths": {
        "https://index.docker.io/v1/": {
            "auth": "c3R...zE2"
        }
    }
}
```

> **Note:** If you use a Docker credentials store, you won't see that `auth` entry but a `credsStore` entry with the name of the store as value.

Create a Secret that holds your authorization token

Create a Secret named `regsecret`:

```
kubectl create secret docker-registry regsecret --docker-server=<your-registry-ser
```

where:

- `<your-registry-server>` is your Private Docker Registry FQDN.
- `<your-name>` is your Docker username.
- `<your-pword>` is your Docker password.
- `<your-email>` is your Docker email.

Understanding your Secret

To understand what's in the Secret you just created, start by viewing the Secret in YAML format:

```
kubectl get secret regsecret --output=yaml
```

The output is similar to this:

```
apiVersion: v1
data:
  .dockercfg: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
kind: Secret
metadata:
  ...
  name: regsecret
  ...
type: kubernetes.io/dockercfg
```

The value of the `.dockercfg` field is a base64 representation of your secret data.

Copy the base64 representation of the secret data into a file named `secret64`.

Important: Make sure there are no line breaks in your `secret64` file.

To understand what is in the `.dockercfg` field, convert the secret data to a readable format:

```
base64 -d secret64
```

The output is similar to this:

```
{"yourprivateregistry.com":{"username":"janedoe","password":"xxxxxxxxxxx","email":
```

Notice that the secret data contains the authorization token from your `config.json` file.

Create a Pod that uses your Secret

Here is a configuration file for a Pod that needs access to your secret data:

private-reg-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
    - name: private-reg-container
      image: <your-private-image>
  imagePullSecrets:
    - name: regsecret
```

Copy the contents of `private-reg-pod.yaml` to your own file named `my-private-reg-pod.yaml`. In your file, replace `<your-private-image>` with the path to an image in a private repository.

Example Docker Hub private image:

```
janedoe/jdoe-private:v1
```

To pull the image from the private repository, Kubernetes needs credentials. The `imagePullSecrets` field in the configuration file specifies that Kubernetes should get the credentials from a Secret named `regsecret`.

Create a Pod that uses your Secret, and verify that the Pod is running:

```
kubectl create -f my-private-reg-pod.yaml
kubectl get pod private-reg
```

What’s next

- Learn more about Secrets.
- Learn more about using a private registry.
- See kubectl create secret docker-registry.
- See Secret

- See the `imagePullSecrets` field of PodSpec.

Configure Liveness and Readiness Probes

This page shows how to configure liveness and readiness probes for Containers.

The kubelet uses liveness probes to know when to restart a Container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Define a liveness command

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations.

In this exercise, you create a Pod that runs a Container based on the `gcr.io/google_containers/busybox` image. Here is the configuration file for the Pod:

exec-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: gcr.io/google_containers/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

In the configuration file, you can see that the Pod has a single Container. The `periodSeconds` field specifies that the kubelet should perform a liveness probe every 5 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 5 second before performing the first probe. To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the Container. If the command succeeds, it returns 0, and the kubelet considers the Container to be alive and healthy. If the command returns a non-zero value, the kubelet kills the Container and restarts it.

When the Container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

For the first 30 seconds of the Container's life, there is a `/tmp/healthy` file. So during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/exec-liveness.
```

Within 30 seconds, view the Pod events:

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

```
FirstSeen    LastSeen    Count   From            SubobjectPath           Type
--------- --------    -----   ----            -------------           --------
24s       24s     1   {default-scheduler }                    Normal      Schedule
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal
23s       23s     1   {kubelet worker0}   spec.containers{liveness}   Normal
```

After 35 seconds, view the Pod events again:

```
kubectl describe pod liveness-exec
```

At the bottom of the output, there are messages indicating that the liveness probes have failed, and the containers have been killed and recreated.

```
FirstSeen LastSeen    Count   From            SubobjectPath           Type
--------- --------    -----   ----            -------------           --------
37s       37s     1   {default-scheduler }                    Normal      Schedule
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal
2s        2s      1   {kubelet worker0}   spec.containers{liveness}   Warning
```

Wait another 30 seconds, and verify that the Container has been restarted:

```
kubectl get pod liveness-exec
```

The output shows that `RESTARTS` has been incremented:

```
NAME            READY     STATUS    RESTARTS   AGE
liveness-exec   1/1       Running   1          1m
```

Define a liveness HTTP request

Another kind of liveness probe uses an HTTP GET request. Here is the configuration file for a Pod that runs a container based on the `gcr.io/google_containers/liveness` image.

http-liveness.yaml

http-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: gcr.io/google_containers/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
          - name: X-Custom-Header
            value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

In the configuration file, you can see that the Pod has a single Container. The `livenessProbe` field specifies that the kubelet should perform a liveness probe every 3 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 3 seconds before performing the first probe. To perform a probe, the kubelet sends an HTTP GET request to the server that is running in the Container and listening on port 8080. If the handler for the server's `/healthz` path returns a success code, the kubelet considers the Container to be alive and healthy. If the handler returns a failure code, the kubelet kills the Container and restarts it.

Any code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

You can see the source code for the server in server.go.

For the first 10 seconds that the Container is alive, the `/healthz` handler returns a status of 200. After that, the handler returns a status of 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

The kubelet starts performing health checks 3 seconds after the Container starts. So the first couple of health checks will succeed. But after 10 seconds, the health checks will fail, and the kubelet will kill and restart the Container.

To try the HTTP liveness check, create a Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/http-liveness.
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the Container has been restarted:

```
kubectl describe pod liveness-http
```

Define a TCP liveness probe

A third type of liveness probe uses a TCP Socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

tcp-liveness-readiness.yaml

tcp-liveness-readiness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: gcr.io/google_containers/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

As you can see, configuration for a TCP check is quite similar to an HTTP check. This example uses both readiness and liveness probes. The kubelet will send the first readiness probe 5 seconds after the container starts. This will attempt to connect to the `goproxy` container on port 8080. If the probe succeeds, the pod will be marked as ready. The kubelet will continue to run this check every 10 seconds.

In addition to the readiness probe, this configuration includes a liveness probe. The kubelet will run the first liveness probe 15 seconds after the container starts. Just like the readiness probe, this will attempt to connect to the `goproxy` container on port 8080. If the liveness probe fails, the container will be restarted.

Use a named port

You can use a named ContainerPort for HTTP or TCP liveness checks:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
```

Define readiness probes

Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup. In such cases, you don't want to kill the application, but you don't want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

Readiness probes are configured similarly to liveness probes. The only difference is that you use the `readinessProbe` field instead of the `livenessProbe` field.

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Configuration for HTTP and TCP readiness probes also remains identical to liveness probes.

Readiness and liveness probes can be used in parallel for the same container. Using both can ensure that traffic does not reach a container that is not ready for it, and that containers are restarted when they fail.

Configure Probes

Probes have a number of fields that you can use to more precisely control the behavior of liveness and readiness checks:

- `initialDelaySeconds` : Number of seconds after the container has started before liveness probes are initiated.
- `periodSeconds` : How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
- `timeoutSeconds` : Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
- `successThreshold` : Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
- `failureThreshold` : Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.

HTTP probes have additional fields that can be set on `httpGet` :

- `host` : Host name to connect to, defaults to the pod IP. You probably want to set "Host" in httpHeaders instead.
- `scheme` : Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.
- `path` : Path to access on the HTTP server.
- `httpHeaders` : Custom headers to set in the request. HTTP allows repeated headers.
- `port` : Name or number of the port to access on the container. Number must be in the range 1 to 65535.

For an HTTP probe, the kubelet sends an HTTP request to the specified path and port to perform the check. The kubelet sends the probe to the container's IP address, unless the address is overridden by the optional `host` field in `httpGet` . If `scheme` field is set to `HTTPS` , the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the `host` field. Here's one scenario where you would set it. Suppose the Container listens on 127.0.0.1 and the Pod's `hostNetwork` field is true. Then `host` , under `httpGet` , should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use `host`, but rather set the `Host` header in `httpHeaders`.

What’s next

- Learn more about Container Probes.

Reference

- Pod
- Container
- Probe

Assign Pods to Nodes

This page shows how to assign a Kubernetes Pod to a particular node in a Kubernetes cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Add a label to a node

1. List the nodes in your cluster:

```
kubectl get nodes
```

The output is similar to this:

```
NAME      STATUS    AGE     VERSION
worker0   Ready     1d      v1.6.0+fff5156
worker1   Ready     1d      v1.6.0+fff5156
worker2   Ready     1d      v1.6.0+fff5156
```

2. Chose one of your nodes, and add a label to it:

   ```
   kubectl label nodes <your-node-name> disktype=ssd
   ```

   where `<your-node-name>` is the name of your chosen node.

3. Verify that your chosen node has a `disktype=ssd` label:

   ```
   kubectl get nodes --show-labels
   ```

   The output is similar to this:

   ```
    NAME      STATUS    AGE     VERSION           LABELS
    worker0   Ready     1d      v1.6.0+fff5156    ...,disktype=ssd,kubernetes.io
    worker1   Ready     1d      v1.6.0+fff5156    ...,kubernetes.io/hostname=worl
    worker2   Ready     1d      v1.6.0+fff5156    ...,kubernetes.io/hostname=worl
   ```

   In the preceding output, you can see that the `worker0` node has a `disktype=ssd` label.

Create a pod that gets scheduled to your chosen node

This pod configuration file describes a pod that has a node selector, `disktype: ssd`. This means that the pod will get scheduled on a node that has a `disktype=ssd` label.

pod.yaml

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

1. Use the configuration file to create a pod that will get scheduled on your chosen node:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/pod.yaml
```

2. Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

```
NAME     READY    STATUS    RESTARTS   AGE    IP           NODE
nginx    1/1      Running   0          13s    10.200.0.4   worker0
```

What's next

Learn more about labels and selectors.

Configure Pod Initialization

This page shows how to use an Init Container to initialize a Pod before an application Container runs.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Create a Pod that has an Init Container

In this exercise you create a Pod that has one application Container and one Init Container. The init container runs to completion before the application container starts.

Here is the configuration file for the Pod:

init-containers.yaml

init-containers.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html
  # These containers are run during pod initialization
  initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://kubernetes.io
    volumeMounts:
    - name: workdir
      mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
```

In the configuration file, you can see that the Pod has a Volume that the init container and the application container share.

The init container mounts the shared Volume at `/work-dir`, and the application container mounts the shared Volume at `/usr/share/nginx/html`. The init container runs the following command and then terminates:

```
wget -O /work-dir/index.html http://kubernetes.io
```

Notice that the init container writes the `index.html` file in the root directory of the nginx server.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/init-container
```

Verify that the nginx container is running:

```
kubectl get pod init-demo
```

The output shows that the nginx container is running:

```
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          43m
```

Get a shell into the nginx container running in the init-demo Pod:

```
kubectl exec -it init-demo -- /bin/bash
```

In your shell, send a GET request to the nginx server:

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

The output shows that nginx is serving the web page that was written by the init container:

```
<!Doctype html>
<html id="home">

<head>
...
"url": "http://kubernetes.io/"}</script>
</head>
<body>
  ...
  <p>Kubernetes is open source giving you the freedom to take advantage ...</p>
  ...
```

What’s next

- Learn more about communicating between Containers running in the same Pod.
- Learn more about Init Containers.
- Learn more about Volumes.
- Learn more about Debugging Init Containers

Attach Handlers to Container Lifecycle Events

This page shows how to attach handlers to Container lifecycle events. Kubernetes supports the postStart and preStop events. Kubernetes sends the postStart event immediately after a Container is started, and it sends the preStop event immediately before the Container is terminated.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Define postStart and preStop handlers

In this exercise, you create a Pod that has one Container. The Container has handlers for the postStart and preStop events.

Here is the configuration file for the Pod:

`lifecycle-events.yaml`

lifecycle-events.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx

    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

In the configuration file, you can see that the postStart command writes a `message` file to the Container's `/usr/share` directory. The preStop command shuts down nginx gracefully. This is helpful if the Container is being terminated because of a failure.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/lifecycle-even
```

Verify that the Container in the Pod is running:

```
kubectl get pod lifecycle-demo
```

Get a shell into the Container running in your Pod:

```
kubectl exec -it lifecycle-demo -- /bin/bash
```

In your shell, verify that the `postStart` handler created the `message` file:

```
root@lifecycle-demo:/# cat /usr/share/message
```

The output shows the text written by the postStart handler:

```
Hello from the postStart handler
```

Discussion

Kubernetes sends the postStart event immediately after the Container is created. There is no guarantee, however, that the postStart handler is called before the Container's entrypoint is called. The postStart handler runs asynchronously relative to the Container's code, but Kubernetes' management of the container blocks until the postStart handler completes. The Container's status is not set to RUNNING until the postStart handler completes.

Kubernetes sends the preStop event immediately before the Container is terminated. Kubernetes' management of the Container blocks until the preStop handler completes, unless the Pod's grace period expires. For more details, see Termination of Pods.

What's next

- Learn more about Container lifecycle hooks.
- Learn more about the lifecycle of a Pod.

Reference

- Lifecycle
- Container
- See `terminationGracePeriodSeconds` in PodSpec

Configure Containers Using a ConfigMap

This page shows you how to configure an application using a ConfigMap. ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes

Use kubectl to create a ConfigMap

Use the `kubectl create configmap` command to create configmaps from directories, files, or literal values:

```
kubectl create configmap <map-name> <data-source>
```

where <map-name> is the name you want to assign to the ConfigMap and <data-source> is the directory, file, or literal value to draw the data from.

The data source corresponds to a key-value pair in the ConfigMap, where

- key = the file name or the key you provided on the command line, and
- value = the file contents or the literal value you provided on the command line.

You can use `kubectl describe` or `kubectl get` to retrieve information about a ConfigMap. The former shows a summary of the ConfigMap, while the latter returns the full contents of the ConfigMap.

Create ConfigMaps from directories

You can use `kubectl create configmap` to create a ConfigMap from multiple files in the same directory.

For example:

```
kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl
```

combines the contents of the `docs/user-guide/configmap/kubectl/` directory

```
ls docs/user-guide/configmap/kubectl/
game.properties
ui.properties
```

into the following ConfigMap:

```
kubectl describe configmaps game-config
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
====
game.properties:        158 bytes
ui.properties:          83 bytes
```

The `game.properties` and `ui.properties` files in the `docs/user-guide/configmap/kubectl/` directory are represented in the `data` section of the ConfigMap.

```
kubectl get configmaps game-config -o yaml
```

```
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

Create ConfigMaps from files

You can use `kubectl create configmap` to create a ConfigMap from an individual file, or from multiple files.

For example,

```
kubectl create configmap game-config-2 --from-file=docs/user-guide/configmap/kubec
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
Name:           game-config-2
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
====
game.properties:        158 bytes
```

You can pass in the `--from-file` argument multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap game-config-2 --from-file=docs/user-guide/configmap/kubec
```

```
kubectl describe configmaps game-config-2
Name:           game-config-2
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
====
game.properties:        158 bytes
ui.properties:          83 bytes
```

Define the key to use when creating a ConfigMap from a file

You can define a key other than the file name to use in the `data` section of your ConfigMap when using the `--from-file` argument:

```
kubectl create configmap game-config-3 --from-file=<my-key-name>=<path-to-file>
```

where `<my-key-name>` is the key you want to use in the ConfigMap and `<path-to-file>` is the location of the data source file you want the key to represent.

For example:

```
kubectl create configmap game-config-3 --from-file=game-special-key=docs/user-guid

kubectl get configmaps game-config-3 -o yaml
```

```
apiVersion: v1
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

Create ConfigMaps from literal values

You can use `kubectl create configmap` with the `--from-literal` argument to define a literal value from the command line:

```
kubectl create configmap special-config --from-literal=special.how=very --from-lit
```

You can pass in multiple key-value pairs. Each pair provided on the command line is represented as a separate entry in the `data` section of the ConfigMap.

```
kubectl get configmaps special-config -o yaml
```

```
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

Understanding ConfigMaps

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. The ConfigMap API resource stores configuration data as key-value pairs. The data can be consumed in pods or provide the configurations for system components such as controllers. ConfigMap is similar to Secrets, but provides a means of working with strings that don't contain sensitive information. Users and system components alike can store configuration data in ConfigMap.

> **Note:** ConfigMaps should reference properties files, not replace them. Think of the ConfigMap as representing something similar to the Linux `/etc` directory and its contents. For example, if you create a Kubernetes Volume from a ConfigMap, each data item in the ConfigMap is represented by an individual file in the volume.

The ConfigMap's `data` field contains the configuration data. As shown in the example below, this can be simple – like individual properties defined using `--from-literal` – or complex – like configuration files or JSON blobs defined using `--from-file`.

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  # example of a simple property defined using --from-literal
  example.property.1: hello
  example.property.2: world
  # example of a complex property defined using --from-file
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

What’s next

- See Using ConfigMap Data in Pods.
- Follow a real world example of Configuring Redis using a ConfigMap.

Use ConfigMap Data in Pods

This page provides a series of usage examples demonstrating how to configure Pods using data stored in ConfigMaps.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda

- Play with Kubernetes

- Create a ConfigMap

Define Pod environment variables using ConfigMap data

Define a Pod environment variable with data from a single ConfigMap

1. Define an environment variable as a key-value pair in a ConfigMap:

```
kubectl create configmap special-config --from-literal=special.how=very
```

2. Assign the `special.how` value defined in the ConfigMap to the `SPECIAL_LEVEL_KEY` environment variable in the Pod specification.

```
kubectl edit pod dapi-test-pod
```

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        # Define the environment variable
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              # The ConfigMap containing the value you want to assign to SPECI
              name: special-config
              # Specify the key associated with the value
              key: special.how
  restartPolicy: Never
```

3. Save the changes to the Pod specification. Now, the Pod’s output includes `SPECIAL_LEVEL_KEY=very`.

Define Pod environment variables with data from multiple ConfigMaps

1. As with the previous example, create the ConfigMaps first.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

2. Define the environment variables in the Pod specification.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: env-config
              key: log_level
  restartPolicy: Never
```

3. Save the changes to the Pod specification. Now, the Pod's output includes `SPECIAL_LEVEL_KEY=very` and `LOG_LEVEL=info`.

Configure all key-value pairs in a ConfigMap as Pod environment variables

Note: This functionality is available to users running Kubernetes v1.6 and later.

1. Create a ConfigMap containing multiple key-value pairs.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

2. Use `envFrom` to define all of the ConfigMap's data as Pod environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - configMapRef:
          name: special-config
  restartPolicy: Never
```

3. Save the changes to the Pod specification. Now, the Pod's output includes `SPECIAL_LEVEL=very` and `SPECIAL_TYPE=charm`.

Use ConfigMap-defined environment variables in Pod commands

You can use ConfigMap-defined environment variables in the `command` section of the Pod specification using the `$(VAR_NAME)` Kubernetes substitution syntax.

For example:

The following Pod specification

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)"
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special_level
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special_type
  restartPolicy: Never
```

produces the following output in the `test-container` container:

```
very charm
```

Add ConfigMap data to a Volume

As explained in Configure Containers Using a ConfigMap, when you create a ConfigMap using `--from-file`, the filename becomes a key stored in the `data` section of the ConfigMap. The file contents become the key's value.

The examples in this section refer to a ConfigMap named special-config, shown below.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.level: very
  special.type: charm
```

Populate a Volume with data stored in a ConfigMap

Add the ConfigMap name under the `volumes` section of the Pod specification. This adds the ConfigMap data to the directory specified as `volumeMounts.mountPath` (in this case, `/etc/config`). The `command` section references the `special.level` item stored in the ConfigMap.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: special-config
  restartPolicy: Never
```

When the pod runs, the command (`"ls /etc/config/"`) produces the output below:

```
special.level
special.type
```

Add ConfigMap data to a specific path in the Volume

Use the `path` field to specify the desired file path for specific ConfigMap items. In this case, the `special.level` item will be mounted in the `config-volume` volume at `/etc/config/keys`.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh","-c","cat /etc/config/keys" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.level
          path: keys
  restartPolicy: Never
```

When the pod runs, the command (`"cat /etc/config/keys"`) produces the output below:

```
very
```

Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. The Secrets user guide explains the syntax.

Mounted ConfigMaps are updated automatically

When a ConfigMap already being consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted ConfigMap is fresh on every periodic sync. However, it is using its local ttl-based cache for getting the current value of the ConfigMap. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the pod can be as long as kubelet sync period + ttl of ConfigMaps cache in kubelet.

Understanding ConfigMaps and Pods

Restrictions

1. You must create a ConfigMap before referencing it in a Pod specification (unless you mark the ConfigMap as "optional"). If you reference a ConfigMap that doesn't exist, the Pod won't start. Likewise, references to keys that don't exist in the ConfigMap will prevent the pod from starting.
2. If you use `envFrom` to define environment variables from ConfigMaps, keys that are considered invalid will be skipped. The pod will be allowed to start, but the invalid names will be recorded in the event log (`InvalidVariableNames`). The log message lists each skipped key. For example:

   ```
   kubectl get events
   LASTSEEN FIRSTSEEN COUNT NAME          KIND  SUBOBJECT  TYPE     REASON
   0s       0s        1     dapi-test-pod Pod              Warning  InvalidEnvir
   ```

3. ConfigMaps reside in a specific namespace. A ConfigMap can only be referenced by pods residing in the same namespace.
4. Kubelet doesn't support the use of ConfigMaps for pods not found on the API server. This includes every pod created using kubectl or indirectly via a replication controller. It does not include pods created via the Kubelet's `--manifest-url` flag, `--config` flag, or the Kubelet REST API.

   Note: These are not commonly-used ways to create pods.

What's next

- Learn more about ConfigMaps.

- Follow a real world example of Configuring Redis using a ConfigMap.

Translate a Docker Compose File to Kubernetes Resources

Kubernetes + Compose = Kompose

What's Kompose? It's a conversion tool for all things compose (namely Docker Compose) to container orchestrators (Kubernetes or OpenShift).

More information can be found our website at http://kompose.io

In three simple steps, we'll take you from Docker Compose to Kubernetes.

1. Take a sample docker-compose.yaml file

```
version: "2"

services:

  redis-master:
    image: gcr.io/google_containers/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v1
    ports:
      - "6379"
    environment:
      - GET_HOSTS_FROM=dns

  frontend:
    image: gcr.io/google-samples/gb-frontend:v4
    ports:
      - "80:80"
    environment:
      - GET_HOSTS_FROM=dns
    labels:
      kompose.service.type: LoadBalancer
```

2. Run `kompose up` in the same directory

```
$ kompose up
We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims
If you need different kind of resources, use the 'kompose convert' and 'kubectl cr

INFO Successfully created Service: redis
INFO Successfully created Service: web
INFO Successfully created Deployment: redis
INFO Successfully created Deployment: web

Your application has been deployed to Kubernetes. You can run 'kubectl get deploym
```

Alternatively, you can run `kompose convert` and deploy with `kubectl`

2.1. Run `kompose convert` in the same directory

```
$ kompose convert
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

2.2. And start it on Kubernetes!

```
$ kubectl create -f frontend-service.yaml,redis-master-service.yaml,redis-slave-se
service "frontend" created
service "redis-master" created
service "redis-slave" created
deployment "frontend" created
deployment "redis-master" created
deployment "redis-slave" created
```

3. View the newly deployed service

Now that your service has been deployed, let’s access it.

If you’re already using `minikube` for your development process:

```
$ minikube service frontend
```

Otherwise, let’s look up what IP your service is using!

```
$ kubectl describe svc frontend
Name:                   frontend
Namespace:              default
Labels:                 service=frontend
Selector:               service=frontend
Type:                   LoadBalancer
IP:                     10.0.0.183
LoadBalancer Ingress:   123.45.67.89
Port:                   80      80/TCP
NodePort:               80      31144/TCP
Endpoints:              172.17.0.4:80
Session Affinity:       None
No events.
```

If you're using a cloud provider, your IP will be listed next to `LoadBalancer Ingress`.

```
$ curl http://123.45.67.89
```

Installation

We have multiple ways to install Kompose. Our prefered method is downloading the binary from the latest GitHub release.

GitHub release

Kompose is released via GitHub on a three-week cycle, you can see all current releases on the GitHub release page.

```
# Linux
curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-lin

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-dar

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-win

chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Alternatively, you can download the tarball.

Go

Installing using `go get` pulls from the master branch with the latest development changes.

```
go get -u github.com/kubernetes/kompose
```

CentOS

Kompose is in EPEL CentOS repository. If you don't have EPEL repository already installed and enabled you can do it by running `sudo yum install epel-release`

If you have EPEL enabled in your system, you can install Kompose like any other package.

```
sudo yum -y install kompose
```

Fedora

Kompose is in Fedora 24, 25 and 26 repositories. You can install it just like any other package.

```
sudo dnf -y install kompose
```

macOS

On macOS you can install latest release via Homebrew:

```
brew install kompose
```

User Guide

- CLI
  - `kompose convert`
  - `kompose up`
  - `kompose down`
- Documentation
  - Build and Push Docker Images
  - Alternative Conversions
  - Labels
  - Restart

- Docker Compose Versions

Kompose has support for two providers: OpenShift and Kubernetes. You can choose a targeted provider using global option `--provider`. If no provider is specified, Kubernetes is set by default.

kompose convert

Kompose supports conversion of V1, V2, and V3 Docker Compose files into Kubernetes and OpenShift objects.

Kubernetes

```
$ kompose --file docker-voting.yml convert
WARN Unsupported key networks - ignoring
WARN Unsupported key build - ignoring
INFO Kubernetes file "worker-svc.yaml" created
INFO Kubernetes file "db-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "result-svc.yaml" created
INFO Kubernetes file "vote-svc.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
INFO Kubernetes file "result-deployment.yaml" created
INFO Kubernetes file "vote-deployment.yaml" created
INFO Kubernetes file "worker-deployment.yaml" created
INFO Kubernetes file "db-deployment.yaml" created

$ ls
db-deployment.yaml  docker-compose.yml         docker-gitlab.yml  redis-deployment
db-svc.yaml         docker-voting.yml          redis-svc.yaml     result-svc.yaml
```

You can also provide multiple docker-compose files at the same time:

```
$ kompose -f docker-compose.yml -f docker-guestbook.yml convert
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "mlbparks-service.yaml" created
INFO Kubernetes file "mongodb-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "mlbparks-deployment.yaml" created
INFO Kubernetes file "mongodb-deployment.yaml" created
INFO Kubernetes file "mongodb-claim0-persistentvolumeclaim.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created

$ ls
mlbparks-deployment.yaml  mongodb-service.yaml                       redis-slave-s
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  redis-master-
frontend-service.yaml     mongodb-deployment.yaml                    redis-slave-d
redis-master-deployment.yaml
```

When multiple docker-compose files are provided the configuration is merged. Any configuration that is common will be over ridden by subsequent file.

OpenShift

```
$ kompose --provider openshift --file docker-voting.yml convert
WARN [worker] Service cannot be created because of missing port.
INFO OpenShift file "vote-service.yaml" created
INFO OpenShift file "db-service.yaml" created
INFO OpenShift file "redis-service.yaml" created
INFO OpenShift file "result-service.yaml" created
INFO OpenShift file "vote-deploymentconfig.yaml" created
INFO OpenShift file "vote-imagestream.yaml" created
INFO OpenShift file "worker-deploymentconfig.yaml" created
INFO OpenShift file "worker-imagestream.yaml" created
INFO OpenShift file "db-deploymentconfig.yaml" created
INFO OpenShift file "db-imagestream.yaml" created
INFO OpenShift file "redis-deploymentconfig.yaml" created
INFO OpenShift file "redis-imagestream.yaml" created
INFO OpenShift file "result-deploymentconfig.yaml" created
INFO OpenShift file "result-imagestream.yaml" created
```

It also supports creating buildconfig for build directive in a service. By default, it uses the remote repo for the current git branch as the source repo, and the current branch as the source branch for the build. You can specify a different source repo and branch using `--build-repo` and `--build-branch` options respectively.

```
$ kompose --provider openshift --file buildconfig/docker-compose.yml convert
WARN [foo] Service cannot be created because of missing port.
INFO OpenShift Buildconfig using git@github.com:rtnpro/kompose.git::master as sour
INFO OpenShift file "foo-deploymentconfig.yaml" created
INFO OpenShift file "foo-imagestream.yaml" created
INFO OpenShift file "foo-buildconfig.yaml" created
```

Note: If you are manually pushing the Openshift artifacts using `oc create -f`, you need to ensure that you push the imagestream artifact before the buildconfig artifact, to workaround this Openshift issue: https://github.com/openshift/origin/issues/4518 .

kompose up

Kompose supports a straightforward way to deploy your "composed" application to Kubernetes or OpenShift via `kompose up`.

Kubernetes

```
$ kompose --file ./examples/docker-guestbook.yml up
We are going to create Kubernetes deployments and services for your Dockerized app
If you need different kind of resources, use the 'kompose convert' and 'kubectl cr

INFO Successfully created service: redis-master
INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created deployment: frontend

Your application has been deployed to Kubernetes. You can run 'kubectl get deploym

$ kubectl get deployment,svc,pods
NAME                                DESIRED       CURRENT       UP-TO-DATE    AVAILA
deploy/frontend                     1             1             1             1
deploy/redis-master                 1             1             1             1
deploy/redis-slave                  1             1             1             1

NAME                                CLUSTER-IP    EXTERNAL-IP   PORT(S)       AGE
svc/frontend                        10.0.174.12   <none>        80/TCP        4m
svc/kubernetes                      10.0.0.1      <none>        443/TCP       13d
svc/redis-master                    10.0.202.43   <none>        6379/TCP      4m
svc/redis-slave                     10.0.1.85     <none>        6379/TCP      4m

NAME                                READY         STATUS        RESTARTS      AGE
po/frontend-2768218532-cs5t5        1/1           Running       0             4m
po/redis-master-1432129712-63jn8    1/1           Running       0             4m
po/redis-slave-2504961300-nve7b     1/1           Running       0             4m
```

Note: - You must have a running Kubernetes cluster with a pre-configured kubectl context. - Only deployments and services are generated and deployed to Kubernetes. If you need different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands instead.

OpenShift

```
$ kompose --file ./examples/docker-guestbook.yml --provider openshift up
We are going to create OpenShift DeploymentConfigs and Services for your Dockerize
If you need different kind of resources, use the 'kompose convert' and 'oc create 

INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created service: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created ImageStream: redis-slave
INFO Successfully created deployment: frontend
INFO Successfully created ImageStream: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created ImageStream: redis-master

Your application has been deployed to OpenShift. You can run 'oc get dc,svc,is' fo

$ oc get dc,svc,is
NAME                REVISION                               DESIRED       CURRENT
dc/frontend         0                                      1             0
dc/redis-master     0                                      1             0
dc/redis-slave      0                                      1             0
NAME                CLUSTER-IP                             EXTERNAL-IP   PORT(S)
svc/frontend        172.30.46.64                           <none>        80/TCP
svc/redis-master    172.30.144.56                          <none>        6379/TCP
svc/redis-slave     172.30.75.245                          <none>        6379/TCP
NAME                DOCKER REPO                            TAGS          UPDATED
is/frontend         172.30.12.200:5000/fff/frontend
is/redis-master     172.30.12.200:5000/fff/redis-master
is/redis-slave      172.30.12.200:5000/fff/redis-slave    v1
```

Note: - You must have a running OpenShift cluster with a pre-configured `oc` context (`oc login`)

kompose down

Once you have deployed "composed" application to Kubernetes, `$ kompose down` will help you to take the application out by deleting its deployments and services. If you need to remove other resources, use the 'kubectl' command.

```
$ kompose --file docker-guestbook.yml down
INFO Successfully deleted service: redis-master
INFO Successfully deleted deployment: redis-master
INFO Successfully deleted service: redis-slave
INFO Successfully deleted deployment: redis-slave
INFO Successfully deleted service: frontend
INFO Successfully deleted deployment: frontend
```

Note: - You must have a running Kubernetes cluster with a pre-configured kubectl context.

Build and Push Docker Images

Kompose supports both building and pushing Docker images. When using the `build` key within your Docker Compose file, your image will:

- Automatically be built with Docker using the `image` key specified within your file
- Be pushed to the correct Docker repository using local credentials (located at `.docker/config`)

Using an example Docker Compose file:

```
version: "2"

services:
    foo:
        build: "./build"
        image: docker.io/foo/bar
```

Using `kompose up` with a `build` key:

```
$ kompose up
INFO Build key detected. Attempting to build and push image 'docker.io/foo/bar'
INFO Building image 'docker.io/foo/bar' from directory 'build'
INFO Image 'docker.io/foo/bar' from directory 'build' built successfully
INFO Pushing image 'foo/bar:latest' to registry 'docker.io'
INFO Attempting authentication credentials 'https://index.docker.io/v1/
INFO Successfully pushed image 'foo/bar:latest' to registry 'docker.io'
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeC

INFO Deploying application in "default" namespace
INFO Successfully created Service: foo
INFO Successfully created Deployment: foo

Your application has been deployed to Kubernetes. You can run 'kubectl get deploym
```

In order to disable the functionality, or choose to use BuildConfig generation (with OpenShift) `--build (local|build-config|none)` can be passed.

```
# Disable building/pushing Docker images
$ kompose up --build none

# Generate Build Config artifacts for OpenShift
$ kompose up --provider openshift --build build-config
```

Alternative Conversions

The default `kompose` transformation will generate Kubernetes Deployments and Services, in yaml format. You have alternative option to generate json with `-j`. Also, you can alternatively generate Replication Controllers objects, Deamon Sets, or Helm charts.

```
$ kompose convert -j
INFO Kubernetes file "redis-svc.json" created
INFO Kubernetes file "web-svc.json" created
INFO Kubernetes file "redis-deployment.json" created
INFO Kubernetes file "web-deployment.json" created
```

The `*-deployment.json` files contain the Deployment objects.

```
$ kompose convert --replication-controller
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-replicationcontroller.yaml" created
INFO Kubernetes file "web-replicationcontroller.yaml" created
```

The `*-replicationcontroller.yaml` files contain the Replication Controller objects. If you want to specify replicas (default is 1), use `--replicas` flag:

```
$ kompose convert --replication-controller --replicas 3
```

```
$ kompose convert --daemon-set
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-daemonset.yaml" created
INFO Kubernetes file "web-daemonset.yaml" created
```

The `*-daemonset.yaml` files contain the Daemon Set objects

If you want to generate a Chart to be used with Helm simply do:

```
$ kompose convert -c
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-deployment.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
chart created in "./docker-compose/"

$ tree docker-compose/
docker-compose
├── Chart.yaml
├── README.md
└── templates
    ├── redis-deployment.yaml
    ├── redis-svc.yaml
    ├── web-deployment.yaml
    └── web-svc.yaml
```

The chart structure is aimed at providing a skeleton for building your Helm charts.

Labels

`kompose` supports Kompose-specific labels within the `docker-compose.yml` file in order to explicitly define a service's behavior upon conversion.

- kompose.service.type defines the type of service to be created.

For example:

```
version: "2"
services:
  nginx:
    image: nginx
    dockerfile: foobar
    build: ./foobar
    cap_add:
      - ALL
    container_name: foobar
    labels:
      kompose.service.type: nodeport
```

- kompose.service.expose defines if the service needs to be made accessible from outside the cluster or not. If the value is set to "true", the provider sets the endpoint automatically, and for any other value, the value is set as the hostname. If multiple ports are defined in a service, the first one is chosen to be the exposed.
  - For the Kubernetes provider, an ingress resource is created and it is assumed that an ingress controller has already been configured.
  - For the OpenShift provider, a route is created.

For example:

```
version: "2"
services:
  web:
    image: tuna/docker-counter23
    ports:
     - "5000:5000"
    links:
     - redis
    labels:
      kompose.service.expose: "counter.example.com"
  redis:
    image: redis:3.0
    ports:
     - "6379"
```

The currently supported options are:

Key	Value
kompose.service.type	nodeport / clusterip / loadbalancer
kompose.service.expose	true / hostname

Note: `kompose.service.type` label should be defined with `ports` only, otherwise `kompose` will fail.

Restart

If you want to create normal pods without controllers you can use `restart` construct of docker-compose to define that. Follow table below to see what heppens on the `restart` value.

`docker-compose` `restart`	object created	Pod `restartPolicy`
`""`	controller object	`Always`
`always`	controller object	`Always`
`on-failure`	Pod	`OnFailure`
`no`	Pod	`Never`

Note: controller object could be `deployment` or `replicationcontroller`, etc.

For e.g. `pival` service will become pod down here. This container calculated value of `pi`.

```
version: '2'

services:
  pival:
    image: perl
    command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    restart: "on-failure"
```

Warning about Deployment Config's

If the Docker Compose file has a volume specified for a service, the Deployment (Kubernetes) or DeploymentConfig (OpenShift) strategy is changed to "Recreate" instead of "RollingUpdate" (default). This is done to avoid multiple instances of a service from accessing a volume at the same time.

If the Docker Compose file has service name with `_` in it (eg. `web_service`), then it will be replaced by `-` and the service name will be renamed accordingly (eg. `web-service`). Kompose does this because "Kubernetes" doesn't allow `_` in object name.

Please note that changing service name might break some `docker-compose` files.

Docker Compose Versions

Kompose supports Docker Compose versions: 1, 2 and 3. We have limited support on versions 2.1 and 3.2 due to their experimental nature.

A full list on compatibility between all three versions is listed in our conversion document including a list of all incompatible Docker Compose keys.

Define a Command and Arguments for a Container

This page shows how to define commands and arguments when you run a container in a Kubernetes Pod.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Define a command and arguments when you create a Pod

When you create a Pod, you can define a command and arguments for the containers that run in the Pod. To define a command, include the `command` field in the configuration file. To define arguments for the command, include the `args` field in the configuration file. The command and arguments that you define cannot be changed after the Pod is created.

The command and arguments that you define in the configuration file override the default command and arguments provided by the container image. If you define args, but do not define a command, the default command is used with your new arguments.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a command and two arguments:

commands.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
  - name: command-demo-container
    image: debian
    command: ["printenv"]
    args: ["HOSTNAME", "KUBERNETES_PORT"]
```

1. Create a Pod based on the YAML configuration file:

   ```
   kubectl create -f https://k8s.io/docs/tasks/inject-data-application/commands.y
   ```

2. List the running Pods:

   ```
   kubectl get pods
   ```

   The output shows that the container that ran in the command-demo Pod has completed.

3. To see the output of the command that ran in the container, view the logs from the Pod:

   ```
   kubectl logs command-demo
   ```

The output shows the values of the HOSTNAME and KUBERNETES_PORT environment variables:

```
command-demo
tcp://10.3.240.1:443
```

Use environment variables to define arguments

In the preceding example, you defined the arguments directly by providing strings. As an alternative to providing strings directly, you can define arguments by using environment variables:

```
env:
- name: MESSAGE
  value: "hello world"
command: ["/bin/echo"]
args: ["$(MESSAGE)"]
```

This means you can define an argument for a Pod using any of the techniques available for defining environment variables, including ConfigMaps and Secrets.

Note: The environment variable appears in parentheses, `"$(VAR)"`. This is required for the variable to be expanded in the `command` or `args` field.

Run a command in a shell

In some cases, you need your command to run in a shell. For example, your command might consist of several commands piped together, or it might be a shell script. To run your command in a shell, wrap it like this:

```
command: ["/bin/sh"]
args: ["-c", "while true; do echo hello; sleep 10;done"]
```

Notes

This table summarizes the field names used by Docker and Kubernetes.

Description	Docker field name	Kubernetes field name
The command run by the container	Entrypoint	command
The arguments passed to the command	Cmd	args

When you override the default Entrypoint and Cmd, these rules apply:

- If you do not supply `command` or `args` for a Container, the defaults defined in the Docker image are used.
- If you supply a `command` but no `args` for a Container, only the supplied `command` is used. The default EntryPoint and the default Cmd defined in the Docker image are ignored.
- If you supply only `args` for a Container, the default Entrypoint defined in the Docker image is run with the `args` that you supplied.
- If you supply a `command` and `args`, the default Entrypoint and the default Cmd defined in the Docker image are ignored. Your `command` is run with your `args`.

Here are some examples:

Image Entrypoint	Image Cmd	Container command	Container args	Command run
`[/ep-1]`	`[foo bar]`	<not set>	<not set>	`[ep-1 foo bar]`
`[/ep-1]`	`[foo bar]`	`[/ep-2]`	<not set>	`[ep-2]`
`[/ep-1]`	`[foo bar]`	<not set>	`[zoo boo]`	`[ep-1 zoo boo]`
`[/ep-1]`	`[foo bar]`	`[/ep-2]`	`[zoo boo]`	`[ep-2 zoo boo]`

What's next

- Learn more about containers and commands.
- Learn more about configuring pods and containers.
- Learn more about running commands in a container.
- See Container.

Define Environment Variables for a Container

This page shows how to define environment variables when you run a container in a Kubernetes Pod.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Define an environment variable for a container

When you create a Pod, you can set environment variables for the containers that run in the Pod. To set environment variables, include the `env` or `envFrom` field in the configuration file.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines an environment variable with name `DEMO_GREETING` and value `"Hello from the environment"`. Here is the configuration file for the Pod:

envars.yaml

envars.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    env:
    - name: DEMO_GREETING
      value: "Hello from the environment"
```

1. Create a Pod based on the YAML configuration file:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/envars.yam
```

2. List the running Pods:

```
kubectl get pods -l purpose=demonstrate-envars
```

The output is similar to this:

```
NAME            READY     STATUS    RESTARTS   AGE
envar-demo      1/1       Running   0          9s
```

3. Get a shell to the container running in your Pod:

```
kubectl exec -it envar-demo -- /bin/bash
```

4. In your shell, run the `printenv` command to list the environment variables.

```
root@envar-demo:/# printenv
```

The output is similar to this:

```
NODE_VERSION=4.4.2
EXAMPLE_SERVICE_PORT_8080_TCP_ADDR=10.3.245.237
HOSTNAME=envar-demo
...
DEMO_GREETING=Hello from the environment
```

5. To exit the shell, enter `exit`.

What's next

- Learn more about environment variables.
- Learn about using secrets as environment variables.
- See EnvVarSource.

Expose Pod Information to Containers Through Environment Variables

This page shows how a Pod can use environment variables to expose information about itself to Containers running in the Pod. Environment variables can expose Pod fields and Container fields.

There are two ways to expose Pod and Container fields to a running Container: environment variables and DownwardAPIVolumeFiles. Together, these two ways of exposing Pod and Container fields are called the *Downward API*.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- DownwardAPIVolumeFiles

Together, these two ways of exposing Pod and Container fields are called the *Downward API*.

Use Pod fields as values for environment variables

In this exercise, you create a Pod that has one Container. Here is the configuration file for the Pod:

dapi-envars-pod.yaml

dapi-envars-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          printenv MY_NODE_NAME MY_POD_NAME MY_POD_NAMESPACE;
          printenv MY_POD_IP MY_POD_SERVICE_ACCOUNT;
          sleep 10;
        done;
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: MY_POD_SERVICE_ACCOUNT
          valueFrom:
            fieldRef:
              fieldPath: spec.serviceAccountName
  restartPolicy: Never
```

In the configuration file, you can see five environment variables. The `env` field is an array of EnvVars. The first element in the array specifies that the `MY_NODE_NAME` environment variable gets its value from the Pod's `spec.nodeName` field. Similarly, the other environment variables get their names from Pod fields.

Note: The fields in this example are Pod fields. They are not fields of the Container in the Pod.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/dapi-envars-po
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs dapi-envars-fieldref
```

The output shows the values of selected environment variables:

```
minikube
dapi-envars-fieldref
default
172.17.0.4
default
```

To see why these values are in the log, look at the `command` and `args` fields in the configuration file. When the Container starts, it writes the values of five environment variables to stdout. It repeats this every ten seconds.

Next, get a shell into the Container that is running in your Pod:

```
kubectl exec -it dapi-envars-fieldref -- sh
```

In your shell, view the environment variables:

```
/# printenv
```

The output shows that certain environment variables have been assigned the values of Pod fields:

```
MY_POD_SERVICE_ACCOUNT=default
...
MY_POD_NAMESPACE=default
MY_POD_IP=172.17.0.4
...
MY_NODE_NAME=minikube
...
MY_POD_NAME=dapi-envars-fieldref
```

Use Container fields as values for environment variables

In the preceding exercise, you used Pod fields as the values for environment variables. In this next exercise, you use Container fields as the values for environment variables. Here is the configuration file for a Pod that has one container:

dapi-envars-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          printenv MY_CPU_REQUEST MY_CPU_LIMIT;
          printenv MY_MEM_REQUEST MY_MEM_LIMIT;
          sleep 10;
        done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.memory
  restartPolicy: Never
```

In the configuration file, you can see four environment variables. The `env` field is an array of EnvVars. The first element in the array specifies that the `MY_CPU_REQUEST` environment variable gets its value from the `requests.cpu` field of a Container named `test-container`. Similarly, the other environment variables get their values from Container fields.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/dapi-envars-co
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs dapi-envars-resourcefieldref
```

The output shows the values of selected environment variables:

```
1
1
33554432
67108864
```

What's next

- Defining Environment Variables for a Container
- PodSpec
- Container
- EnvVar
- EnvVarSource

- ObjectFieldSelector
- ResourceFieldSelector

Expose Pod Information to Containers Through Files

This page shows how a Pod can use a DownwardAPIVolumeFile to expose information about itself to Containers running in the Pod. A DownwardAPIVolumeFile can expose Pod fields and Container fields.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- DownwardAPIVolumeFiles

Together, these two ways of exposing Pod and Container fields are called the *Downward API*.

Store Pod fields

In this exercise, you create a Pod that has one Container. Here is the configuration file for the Pod:

dapi-volume.yaml

dapi-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c"]
      args:
      - while true; do
          if [[ -e /etc/labels ]]; then
            echo -en '\n\n'; cat /etc/labels; fi;
          if [[ -e /etc/annotations ]]; then
            echo -en '\n\n'; cat /etc/annotations; fi;
          sleep 5;
        done;
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

In the configuration file, you can see that the Pod has a `downwardAPI` Volume, and the Container mounts the Volume at `/etc`.

Look at the `items` array under `downwardAPI`. Each element of the array is a DownwardAPIVolumeFile. The first element specifies that the value of the Pod's `metadata.labels` field should be stored in a file named `labels`. The second element specifies that the value of the Pod's `annotations` field should be stored in a file named `annotations`.

> **Note:** The fields in this example are Pod fields. They are not fields of the Container in the Pod.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/dapi-volume.ya
```

Verify that Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs kubernetes-downwardapi-volume-example
```

The output shows the contents of the `labels` file and the `annotations` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"

build="two"
builder="john-doe"
```

Get a shell into the Container that is running in your Pod:

```
kubectl exec -it kubernetes-downwardapi-volume-example -- sh
```

In your shell, view the `labels` file:

```
/# cat /etc/labels
```

The output shows that all of the Pod's labels have been written to the `labels` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"
```

Similarly, view the `annotations` file:

```
/# cat /etc/annotations
```

View the files in the `/etc` directory:

```
/# ls -laR /etc
```

In the output, you can see that the `labels` and `annotations` files are in a temporary subdirectory: in this example, `..2982_06_02_21_47_53.299460680`. In the `/etc` directory, `..data` is a symbolic link to the temporary subdirectory. Also in the `/etc` directory, `labels` and `annotations` are symbolic links.

```
drwxr-xr-x  ... Feb 6 21:47 ..2982_06_02_21_47_53.299460680
lrwxrwxrwx  ... Feb 6 21:47 ..data -> ..2982_06_02_21_47_53.299460680
lrwxrwxrwx  ... Feb 6 21:47 annotations -> ..data/annotations
lrwxrwxrwx  ... Feb 6 21:47 labels -> ..data/labels

/etc/..2982_06_02_21_47_53.299460680:
total 8
-rw-r--r--  ... Feb  6 21:47 annotations
-rw-r--r--  ... Feb  6 21:47 labels
```

Using symbolic links enables dynamic atomic refresh of the metadata; updates are written to a new temporary directory, and the `..data` symlink is updated atomically using rename(2).

Exit the shell:

```
/# exit
```

Store Container fields

The preceding exercise, you stored Pod fields in a DownwardAPIVolumeFile. In this next exercise, you store Container fields. Here is the configuration file for a Pod that has one Container:

dapi-volume-resources.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          if [[ -e /etc/cpu_limit ]]; then
            echo -en '\n'; cat /etc/cpu_limit; fi;
          if [[ -e /etc/cpu_request ]]; then
            echo -en '\n'; cat /etc/cpu_request; fi;
          if [[ -e /etc/mem_limit ]]; then
            echo -en '\n'; cat /etc/mem_limit; fi;
          if [[ -e /etc/mem_request ]]; then
            echo -en '\n'; cat /etc/mem_request; fi;
          sleep 5;
        done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
```

In the configuration file, you can see that the Pod has a `downwardAPI` Volume, and the Container mounts the Volume at `/etc`.

Look at the `items` array under `downwardAPI`. Each element of the array is a DownwardAPIVolumeFile.

The first element specifies that in the Container named `client-container`, the value of the `limits.cpu` field should be stored in a file named `cpu_limit`.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/dapi-volume-re
```

Get a shell into the Container that is running in your Pod:

```
kubectl exec -it kubernetes-downwardapi-volume-example-2 -- sh
```

In your shell, view the `cpu_limit` file:

```
/# cat /etc/cpu_limit
```

You can use similar commands to view the `cpu_request`, `mem_limit` and `mem_request` files.

Capabilities of the Downward API

The following information is available to Containers through environment variables and DownwardAPIVolumeFiles:

- The Node's name
- The Node's IP
- The Pod's name
- The Pod's namespace
- The Pod's IP address
- The Pod's service account name
- The Pod's UID
- A Container's CPU limit
- A Container's CPU request
- A Container's memory limit
- A Container's memory request

In addition, the following information is available through DownwardAPIVolumeFiles.

- The Pod's labels
- The Pod's annotations

Note: If CPU and memory limits are not specified for a Container, the Downward API defaults to the node allocatable value for CPU and memory.

Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. For more information, see Secrets.

Motivation for the Downward API

It is sometimes useful for a Container to have information about itself, without being overly coupled to Kubernetes. The Downward API allows containers to consume information about themselves or the cluster without using the Kubernetes client or API server.

An example is an existing application that assumes a particular well-known environment variable holds a unique identifier. One possibility is to wrap the application, but that is tedious and error prone, and it violates the goal of low coupling. A better option would be to use the Pod's name as an identifier, and inject the Pod's name into the well-known environment variable.

What's next

- PodSpec
- Volume
- DownwardAPIVolumeSource
- DownwardAPIVolumeFile
- ResourceFieldSelector

Distribute Credentials Securely Using Secrets

This page shows how to securely inject sensitive data, such as passwords and encryption keys, into Pods.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Convert your secret data to a base-64 representation

Suppose you want to have two pieces of secret data: a username `my-app` and a password `39528$vdg7Jb`. First, use Base64 encoding to convert your username and password to a base-64 representation. Here's a Linux example:

```
echo -n 'my-app' | base64
echo -n '39528$vdg7Jb' | base64
```

The output shows that the base-64 representation of your username is `bXktYXBw`, and the base-64 representation of your password is `Mzk1MjgkdmRnN0pi`.

Create a Secret

Here is a configuration file you can use to create a Secret that holds your username and password:

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBwCg==
  password: Mzk1MjgkdmRnN0piCg==
```

1. Create the Secret

```
kubectl create -f secret.yaml
```

Note: If you want to skip the Base64 encoding step, you can create a Secret by using the `kubectl create secret` command:

```
kubectl create secret generic test-secret --from-literal=username='my-app' --f
```

2. View information about the Secret:

```
kubectl get secret test-secret
```

Output:

```
NAME          TYPE      DATA      AGE
test-secret   Opaque    2         1m
```

3. View more detailed information about the Secret:

```
kubectl describe secret test-secret
```

Output:

```
Name:       test-secret
Namespace:  default
Labels:     <none>
Annotations:    <none>

Type:   Opaque

Data
====
password:   13 bytes
username:   7 bytes
```

Create a Pod that has access to the secret data through a Volume

Here is a configuration file you can use to create a Pod:

secret-pod.yaml

secret-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
          # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/secret-volume
  # The secret data is exposed to Containers in the Pod through a Volume.
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

1. Create the Pod:

   ```
   kubectl create -f secret-pod.yaml
   ```

2. Verify that your Pod is running:

   ```
   kubectl get pod secret-test-pod
   ```

   Output:

   ```
   NAME              READY     STATUS    RESTARTS   AGE
   secret-test-pod   1/1       Running   0          42m
   ```

3. Get a shell into the Container that is running in your Pod:

   ```
   kubectl exec -it secret-test-pod -- /bin/bash
   ```

4. The secret data is exposed to the Container through a Volume mounted under `/etc/secret-volume`. In your shell, go to the directory where the secret data is exposed:

   ```
   root@secret-test-pod:/# cd /etc/secret-volume
   ```

5. In your shell, list the files in the `/etc/secret-volume` directory:

   ```
   root@secret-test-pod:/etc/secret-volume# ls
   ```

   The output shows two files, one for each piece of secret data:

   ```
   password username
   ```

6. In your shell, display the contents of the `username` and `password` files:

   ```
   root@secret-test-pod:/etc/secret-volume# cat username; echo; cat password; ec
   ```

   The output is your username and password:

   ```
   my-app
   39528$vdg7Jb
   ```

Create a Pod that has access to the secret data through environment variables

Here is a configuration file you can use to create a Pod:

secret-envars-pod.yaml

secret-envars-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-envars-test-pod
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: test-secret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: test-secret
          key: password
```

1. Create the Pod:

   ```
   kubectl create -f secret-envars-pod.yaml
   ```

2. Verify that your Pod is running:

   ```
   kubectl get pod secret-envars-test-pod
   ```

   Output:

   ```
   NAME                     READY     STATUS    RESTARTS   AGE
   secret-envars-test-pod   1/1       Running   0          4m
   ```

3. Get a shell into the Container that is running in your Pod:

   ```
   kubectl exec -it secret-envars-test-pod -- /bin/bash
   ```

4. In your shell, display the environment variables:

```
root@secret-envars-test-pod:/# printenv
```

The output includes your username and password:

```
...
SECRET_USERNAME=my-app
...
SECRET_PASSWORD=39528$vdg7Jb
```

What's next

- Learn more about Secrets.
- Learn about Volumes.

Reference

- Secret
- Volume
- Pod

Inject Information into Pods Using a PodPreset

You can use a `podpreset` object to inject certain information into pods at creation time. This information can include secrets, volumes, volume mounts, and environment variables.

See PodPreset proposal for more information.

What is a Pod Preset?

A *Pod Preset* is an API resource that you can use to inject additional runtime requirements into a Pod at creation time. You use label selectors to specify the Pods to which a given Pod Preset applies. Check out more information on label selectors.

Using a Pod Preset allows pod template authors to not have to explicitly set information for every pod. This way, authors of pod templates consuming a specific service do not need to know all the details about that service.

Admission Control

Admission control is how Kubernetes applies Pod Presets to incoming pod creation requests. When a pod creation request occurs, the system does the following:

1. Retrieve all `PodPresets` available for use.
2. Match the label selector of the `PodPreset` to the pod being created.
3. Attempt to merge the various defined resources for the `PodPreset` into the Pod being created.
4. On error, throw an event documenting the merge error on the pod, and create the pod *without* any injected resources from the `PodPreset`.

Behavior

When a `PodPreset` is applied to one or more Pods, Kubernetes modifies the pod spec. For changes to `Env`, `EnvFrom`, and `VolumeMounts`, Kubernetes modifies the container spec for all containers in the Pod; for changes to Volume, Kubernetes modifies the Pod Spec.

Kubernetes annotates the resulting modified pod spec to show that it was modified by a `PodPreset`. The annotation is of the form

```
podpreset.admission.kubernetes.io/podpreset-<pod-preset name>": "<resource version>"
```

.

Enable Pod Preset

In order to use Pod Presets in your cluster you must ensure the following

1. You have enabled the api type `settings.k8s.io/v1alpha1/podpreset`
2. You have enabled the admission controller `PodPreset`
3. You have defined your pod presets

Disable Pod Preset for a pod

There may be instances where you wish for a pod to not be altered by any pod preset mutations. For these events, one can add an annotation in the pod spec of the form: `podpreset.admission.kubernetes.io/exclude: "true"`.

Create a Pod Preset

Simple Pod Spec Example

This is a simple example to show how a Pod spec is modified by the Pod Preset.

User submitted pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

Example Pod Preset:

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Pod spec after admission controller:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Pod Spec with **ConfigMap** Example

This is an example to show how a Pod spec is modified by the Pod Preset that defines a `ConfigMap` for Environment Variables.

User submitted pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

User submitted `ConfigMap`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: etcd-env-config
data:
  number_of_members: "1"
  initial_cluster_state: new
  initial_cluster_token: DUMMY_ETCD_INITIAL_CLUSTER_TOKEN
  discovery_token: DUMMY_ETCD_DISCOVERY_TOKEN
  discovery_url: http://etcd_discovery:2379
  etcdctl_peers: http://etcd:2379
  duplicate_key: FROM_CONFIG_MAP
  REPLACE_ME: "a value"
```

Example Pod Preset:

```yaml
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: 6379
    - name: duplicate_key
      value: FROM_ENV
    - name: expansion
      value: $(REPLACE_ME)
  envFrom:
    - configMapRef:
        name: etcd-env-config
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: /etc/app/config.json
      readOnly: true
      name: secret-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
         secretName: config-details
```

Pod spec after admission controller:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/app/config.json
          readOnly: true
          name: secret-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
        - name: duplicate_key
          value: FROM_ENV
        - name: expansion
          value: $(REPLACE_ME)
      envFrom:
        - configMapRef:
          name: etcd-env-config
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
         secretName: config-details
```

ReplicaSet with Pod Spec Example

The following example shows that only the pod spec is modified by the Pod Preset.

User submitted ReplicaSet:

```
apiVersion: settings.k8s.io/v1alpha1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
          - name: GET_HOSTS_FROM
            value: dns
        ports:
          - containerPort: 80
```

Example Pod Preset:

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      tier: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Pod spec after admission controller:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: guestbook
    tier: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
spec:
  containers:
  - name: php-redis
    image: gcr.io/google_samples/gb-frontend:v3
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
    env:
    - name: GET_HOSTS_FROM
      value: dns
    - name: DB_PORT
      value: "6379"
    ports:
    - containerPort: 80
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Multiple PodPreset Example

This is an example to show how a Pod spec is modified by multiple Pod Injection Policies.

User submitted pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

Example Pod Preset:

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Another Pod Preset:

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: proxy
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  volumeMounts:
    - mountPath: /etc/proxy/configs
      name: proxy-volume
  volumes:
    - name: proxy-volume
      emptyDir: {}
```

Pod spec after admission controller:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
    podpreset.admission.kubernetes.io/podpreset-proxy: "resource version"
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/proxy/configs
          name: proxy-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: proxy-volume
      emptyDir: {}
```

Conflict Example

This is an example to show how a Pod spec is not modified by the Pod Preset when there is a conflict.

User submitted pod spec:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
  volumes:
    - name: cache-volume
      emptyDir: {}
        - containerPort: 80
```

Example Pod Preset:

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
  namespace: myns
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: other-volume
  volumes:
    - name: other-volume
      emptyDir: {}
```

Pod spec after admission controller will not change because of the conflict:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
  volumes:
    - name: cache-volume
      emptyDir: {}
```

If we run `kubectl describe...` we can see the event:

```
$ kubectl describe ...
....
Events:
  FirstSeen                 LastSeen                 Count   From                       Subobj
  Tue, 07 Feb 2017 16:56:12 -0700   Tue, 07 Feb 2017 16:56:12 -0700 1   {podpreset
```

Deleting a Pod Preset

Once you don't need a pod preset anymore, you can delete it with `kubectl`:

```
$ kubectl delete podpreset allow-database
podpreset "allow-database" deleted
```

Update API Objects in Place Using kubectl patch

This task shows how to use `kubectl patch` to update an API object in place. The exercises in this task demonstrate a strategic merge patch and a JSON merge patch.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Use a strategic merge patch to update a Deployment

Here's the configuration file for a Deployment that has two replicas. Each replica is a Pod that has one container:

deployment-patch-demo.yaml

deployment-patch-demo.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: patch-demo
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: patch-demo-ctr
        image: nginx
```

Create the Deployment:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/deployment-patch-demo.
```

View the Pods associated with your Deployment:

```
kubectl get pods
```

The output shows that the Deployment has two Pods. The `1/1` indicates that each Pod has one container:

```
NAME                        READY     STATUS    RESTARTS   AGE
patch-demo-28633765-670qr   1/1       Running   0          23s
patch-demo-28633765-j5qs3   1/1       Running   0          23s
```

Make a note of the names of the running Pods. Later, you will see that these Pods get terminated and replaced by new ones.

At this point, each Pod has one Container that runs the nginx image. Now suppose you want each Pod to have two containers: one that runs nginx and one that runs redis.

Create a file named `patch-file.yaml` that has this content:

```
spec:
  template:
    spec:
      containers:
      - name: patch-demo-ctr-2
        image: redis
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has two Containers:

```
containers:
- image: redis
  imagePullPolicy: Always
  name: patch-demo-ctr-2
  ...
- image: nginx
  imagePullPolicy: Always
  name: patch-demo-ctr
  ...
```

View the Pods associated with your patched Deployment:

```
kubectl get pods
```

The output shows that the running Pods have different names from the Pods that were running previously. The Deployment terminated the old Pods and created two new Pods that comply with the updated Deployment spec. The `2/2` indicates that each Pod has two Containers:

```
NAME                         READY     STATUS    RESTARTS   AGE
patch-demo-1081991389-2wrn5   2/2       Running   0          1m
patch-demo-1081991389-jmg7b   2/2       Running   0          1m
```

Take a closer look at one of the patch-demo Pods:

```
kubectl get pod <your-pod-name> --output yaml
```

The output shows that the Pod has two Containers: one running nginx and one running redis:

```
containers:
- image: redis
  ...
- image: nginx
  ...
```

Notes on the strategic merge patch

With a patch, you do not have to specify an entire object; you specify only the portion of the object that you want to change. For example, in the preceding exercise, you specified one Container in the `containers` list in a `PodSpec`.

The patch you did in the preceding exercise is called a *strategic merge patch*. With a strategic merge patch, you can update a list by specifying only the elements that you want to add to the list. The existing list elements remain, and the new elements are merged with the existing elements. In the preceding exercise, the resulting `containers` list has both the original nginx Container and the new redis Container.

Use a JSON merge patch to update a Deployment

A strategic merge patch is different from a JSON merge patch. With a JSON merge patch, if you want to update a list, you have to specify the entire new list. And the new list completely replaces the existing list.

The **`kubectl patch`** command has a **`type`** parameter that you can set to one of these values:

Parameter value	Merge type
json	JSON Patch, RFC 6902
merge	JSON Merge Patch, RFC 7386
strategic	Strategic merge patch

For a comparison of JSON patch and JSON merge patch, see JSON Patch and JSON Merge Patch.

The default value for the **`type`** parameter is **`strategic`**. So in the preceding exercise, you did a strategic merge patch.

Next, do a JSON merge patch on your same Deployment. Create a file named **`patch-file-2.yaml`** that has this content:

```
spec:
  template:
    spec:
      containers:
      - name: patch-demo-ctr-3
        image: gcr.io/google-samples/node-hello:1.0
```

In your patch command, set **`type`** to **`merge`**:

```
kubectl patch deployment patch-demo --type merge --patch "$(cat patch-file-2.yaml)
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The **`containers`** list that you specified in the patch has only one Container. The output shows that your list of one Container replaced the existing **`containers`** list.

```
spec:
  containers:
  - image: gcr.io/google-samples/node-hello:1.0
    ...
    name: patch-demo-ctr-3
```

List the running Pods:

```
kubectl get pods
```

In the output, you can see that the existing Pods were terminated, and new Pods were created. The `1/1` indicates that each new Pod is running only one Container.

```
NAME                          READY     STATUS    RESTARTS   AGE
patch-demo-1307768864-69308   1/1       Running   0          1m
patch-demo-1307768864-c86dc   1/1       Running   0          1m
```

Alternate forms of the kubectl patch command

The `kubectl patch` command takes YAML or JSON. It can take the patch as a file or directly on the command line.

Create a file named `patch-file.json` that has this content:

```
{
   "spec": {
      "template": {
         "spec": {
            "containers": [
               {
                  "name": "patch-demo-ctr-2",
                  "image": "redis"
               }
            ]
         }
      }
   }
}
```

The following commands are equivalent:

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"
kubectl patch deployment patch-demo --patch $'spec:\n template:\n  spec:\n   conta

kubectl patch deployment patch-demo --patch "$(cat patch-file.json)"
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spec": {"cont
```

Summary

In this exercise, you `kubectl patch` to change the live configuration of a Deployment object. You did not change the configuration file that you originally used to create the Deployment object. Other commands for updating API objects include kubectl annotate, kubectl edit, kubectl replace, kubectl scale, kubectl update. and kubectl apply.

What's next

- Kubernetes Object Management
- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Declarative Management of Kubernetes Objects Using Configuration Files

Upgrade from PetSets to StatefulSets

This page shows how to upgrade from PetSets (Kubernetes version 1.3 or 1.4) to *StatefulSets* (Kubernetes version 1.5 or later).

Before you begin

- If you don't have PetSets in your current cluster, or you don't plan to upgrade your master to Kubernetes 1.5 or later, you can skip this task.

Differences between alpha PetSets and beta StatefulSets

PetSet was introduced as an alpha resource in Kubernetes release 1.3, and was renamed to StatefulSet as a beta resource in 1.5. Here are some notable changes:

- **StatefulSet is the new PetSet**: PetSet is no longer available in Kubernetes release 1.5 or later. It becomes beta StatefulSet. To understand why the name was changed, see this discussion thread.

- **StatefulSet guards against split brain**: StatefulSets guarantee at most one Pod for a given ordinal index can be running anywhere in a cluster, to guard against split brain scenarios with distributed applications. *TODO: Link to doc about fencing.*
- **Flipped debug annotation behavior**: The default value of the debug annotation (`pod.alpha.kubernetes.io/initialized`) is `true` in 1.5 through 1.7. The annotation is completely ignored in 1.8 and above, which always behave as if it were `true`.

  The absence of this annotation will pause PetSet operations, but will NOT pause StatefulSet operations. In most cases, you no longer need this annotation in your StatefulSet manifests.

Upgrading from PetSets to StatefulSets

Note that these steps need to be done in the specified order. You **should NOT upgrade your Kubernetes master, nodes, or `kubectl` to Kubernetes version 1.5 or later**, until told to do so.

Find all PetSets and their manifests

First, find all existing PetSets in your cluster:

```
kubectl get petsets --all-namespaces
```

If you don't find any existing PetSets, you can safely upgrade your cluster to Kubernetes version 1.5 or later.

If you find existing PetSets and you have all their manifests at hand, you can continue to the next step to prepare StatefulSet manifests.

Otherwise, you need to save their manifests so that you can recreate them as StatefulSets later. Here's an example command for you to save all existing PetSets as one file.

```
# Save all existing PetSets in all namespaces into a single file. Only needed when
kubectl get petsets --all-namespaces -o yaml > all-petsets.yaml
```

Prepare StatefulSet manifests

Now, for every PetSet manifest you have, prepare a corresponding StatefulSet manifest:

1. Change `apiVersion` from `apps/v1alpha1` to `apps/v1beta1`.
2. Change `kind` from `PetSet` to `StatefulSet`.
3. If you have the debug hook annotation `pod.alpha.kubernetes.io/initialized` set to `true`, you can remove it because it's redundant. If you don't have this annotation or have it set to `false`, be aware that StatefulSet operations might resume after the upgrade.

   If you are upgrading to 1.6 or 1.7, you can set the annotation explicitly to `false` to maintain the paused behavior. If you are upgrading to 1.8 or above, there's no longer any debug annotation to pause StatefulSets.

It's recommended that you keep both PetSet manifests and StatefulSet manifests, so that you can safely roll back and recreate your PetSets, if you decide not to upgrade your cluster.

Delete all PetSets without cascading

If you find existing PetSets in your cluster in the previous step, you need to delete all PetSets *without cascading*. You can do this from `kubectl` with `--cascade=false`. Note that if the flag isn't set, **cascading deletion will be performed by default**, and all Pods managed by your PetSets will be gone.

Delete those PetSets by specifying file names. This only works when the files contain only PetSets, but not other resources such as Services:

```
# Delete all existing PetSets without cascading
# Note that <pet-set-file> should only contain PetSets that you want to delete, bu
kubectl delete -f <pet-set-file> --cascade=false
```

Alternatively, delete them by specifying resource names:

```
# Alternatively, delete them by name and namespace without cascading
kubectl delete petsets <pet-set-name> -n=<pet-set-namespace> --cascade=false
```

Make sure you've deleted all PetSets in the system:

```
# Get all PetSets again to make sure you deleted them all
# This should return nothing
kubectl get petsets --all-namespaces
```

At this moment, you've deleted all PetSets in your cluster, but not their Pods, Persistent Volumes, or Persistent Volume Claims. However, since the Pods are not managed by PetSets anymore, they will be vulnerable to node failures until you finish the master upgrade and recreate StatefulSets.

Upgrade your master to Kubernetes version 1.5 or later

Now, you can upgrade your Kubernetes master to Kubernetes version 1.5 or later. Note that **you should NOT upgrade Nodes at this time**, because the Pods (that were once managed by PetSets) are now vulnerable to node failures.

Upgrade kubectl to Kubernetes version 1.5 or later

Upgrade `kubectl` to Kubernetes version 1.5 or later, following the steps for installing and setting up kubectl.

Create StatefulSets

Make sure you have both master and `kubectl` upgraded to Kubernetes version 1.5 or later before continuing:

```
kubectl version
```

The output is similar to this:

```
Client Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.0", GitCommit:
Server Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.0", GitCommit:
```

If both `Client Version` (`kubectl` version) and `Server Version` (master version) are 1.5 or later, you are good to go.

Create StatefulSets to adopt the Pods belonging to the deleted PetSets with the StatefulSet manifests generated in the previous step:

```
kubectl create -f <stateful-set-file>
```

Make sure all StatefulSets are created and running as expected in the newly-upgraded cluster:

```
kubectl get statefulsets --all-namespaces
```

Upgrade nodes to Kubernetes version 1.5 or later (optional)

You can now upgrade Kubernetes nodes to Kubernetes version 1.5 or later. This step is optional, but needs to be done after all StatefulSets are created to adopt PetSets' Pods.

You should be running Node version >= 1.1.0 to run StatefulSets safely. Older versions do not support features which allow the StatefulSet to guarantee that at any time, there is **at most** one Pod with a given identity running in a cluster.

What's next

Learn more about scaling a StatefulSet.

Scale a StatefulSet

This page shows how to scale a StatefulSet.

Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later.
- **Not all stateful applications scale nicely.** You need to understand your StatefulSets well before continuing. If you're unsure, remember that it might not be safe to scale your StatefulSets.
- You should perform scaling only when you're sure that your stateful application cluster is completely healthy.

Use kubectl to scale StatefulSets

Make sure you have `kubectl` upgraded to Kubernetes version 1.5 or later before continuing. If you're unsure, run `kubectl version` and check `Client Version` for which kubectl you're using.

kubectl scale

First, find the StatefulSet you want to scale. Remember, you need to first understand if you can scale it or not.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replicas>
```

Alternative: **kubectl apply** / **kubectl edit** / **kubectl patch**

Alternatively, you can do in-place updates on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply` or `kubectl create --save-config`, update `.spec.replicas` of the StatefulSet manifests, and then do a `kubectl apply`:

```
kubectl apply -f <stateful-set-file-updated>
```

Otherwise, edit that field with `kubectl edit`:

```
kubectl edit statefulsets <stateful-set-name>
```

Or use `kubectl patch`:

```
kubectl patch statefulsets <stateful-set-name> -p '{"spec":{"replicas":<new-replic
```

Troubleshooting

Scaling down doesn't work right

You cannot scale down a StatefulSet when any of the stateful Pods it manages is unhealthy. Scaling down only takes place after those stateful Pods become running and ready.

With a StatefulSet of size > 1, if there is an unhealthy Pod, there is no way for Kubernetes to know (yet) if it is due to a permanent fault or a transient one (upgrade/maintenance/node reboot). If the Pod is unhealthy due to a permanent fault, scaling without correcting the fault may lead to a state where the StatefulSet membership drops below a certain minimum number of “replicas” that are needed to function correctly. This may cause your StatefulSet to become unavailable.

If the Pod is unhealthy due to a transient fault and the Pod might become available again, the transient error may interfere with your scale-up/scale-down operation. Some distributed databases have issues when nodes join and leave at the same time. It is better to reason about scaling operations at the application level in these cases, and perform scaling only when you’re sure that your stateful application cluster is completely healthy.

What’s next

Learn more about deleting a StatefulSet.

Delete a Stateful Set

This task shows you how to delete a StatefulSet.

Before you begin

- This task assumes you have an application running on your cluster represented by a StatefulSet.

Deleting a StatefulSet

You can delete a StatefulSet in the same way you delete other resources in Kubernetes: use the `kubectl delete` command, and specify the StatefulSet either by file or by name.

```
kubectl delete -f <file.yaml>
```

```
kubectl delete statefulsets <statefulset-name>
```

You may need to delete the associated headless service separately after the StatefulSet itself is deleted.

```
kubectl delete service <service-name>
```

Deleting a StatefulSet through kubectl will scale it down to 0, thereby deleting all pods that are a part of it. If you want to delete just the StatefulSet and not the pods, use `--cascade=false` .

```
kubectl delete -f <file.yaml> --cascade=false
```

By passing `--cascade=false` to `kubectl delete`, the Pods managed by the StatefulSet are left behind even after the StatefulSet object itself is deleted. If the pods have a label `app=myapp`, you can then delete them as follows:

```
kubectl delete pods -l app=myapp
```

Persistent Volumes

Deleting the Pods in a StatefulSet will not delete the associated volumes. This is to ensure that you have the chance to copy data off the volume before deleting it. Deleting the PVC after the pods have left the terminating state might trigger deletion of the backing Persistent Volumes depending on the storage class and reclaim policy. You should never assume ability to access a volume after claim deletion.

Note: Use caution when deleting a PVC, as it may lead to data loss.

Complete deletion of a StatefulSet

To simply delete everything in a StatefulSet, including the associated pods, you can run a series of commands similar to the following:

```
grace=$(kubectl get pods <stateful-set-pod> --template '{{.spec.terminationGracePe
kubectl delete statefulset -l app=myapp
sleep $grace
kubectl delete pvc -l app=myapp
```

In the example above, the Pods have the label `app=myapp`; substitute your own label as appropriate.

Force deletion of StatefulSet pods

If you find that some pods in your StatefulSet are stuck in the 'Terminating' or 'Unknown' states for an extended period of time, you may need to manually intervene to forcefully delete the pods from the apiserver. This is a potentially dangerous task. Refer to Deleting StatefulSet Pods for details.

What’s next

Learn more about force deleting StatefulSet Pods.

Force Delete StatefulSet Pods

This page shows how to delete Pods which are part of a stateful set, and explains the considerations to keep in mind when doing so.

Before you begin

- This is a fairly advanced task and has the potential to violate some of the properties inherent to StatefulSet.
- Before proceeding, make yourself familiar with the considerations enumerated below.

StatefulSet considerations

In normal operation of a StatefulSet, there is **never** a need to force delete a StatefulSet Pod. The StatefulSet controller is responsible for creating, scaling and deleting members of the StatefulSet. It tries to ensure that the specified number of Pods from ordinal 0 through N-1 are alive and ready. StatefulSet ensures that, at any time, there is at most one Pod with a given identity running in a cluster. This is referred to as *at most one* semantics provided by a StatefulSet.

Manual force deletion should be undertaken with caution, as it has the potential to violate the at most one semantics inherent to StatefulSet. StatefulSets may be used to run distributed and clustered applications which have a need for a stable network identity and stable storage. These applications often have configuration which relies on an ensemble of a fixed number of members with fixed identities. Having multiple members with the same identity can be disastrous and may lead to data loss (e.g. split brain scenario in quorum-based systems).

Delete Pods

You can perform a graceful pod deletion with the following command:

```
kubectl delete pods <pod>
```

For the above to lead to graceful termination, the Pod **must not** specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. The practice of setting a `pod.Spec.TerminationGracePeriodSeconds` of 0 seconds is unsafe and strongly discouraged for StatefulSet Pods. Graceful deletion is safe and will ensure that the Pod shuts down gracefully before the kubelet deletes the name from the apiserver.

Kubernetes (versions 1.5 or newer) will not delete Pods just because a Node is unreachable. The Pods running on an unreachable Node enter the 'Terminating' or 'Unknown' state after a timeout. Pods may also enter these states when the user attempts graceful deletion of a Pod on an unreachable Node. The only ways in which a Pod in such a state can be removed from the apiserver are as follows:

- The Node object is deleted (either by you, or by the Node Controller).
- The kubelet on the unresponsive Node starts responding, kills the Pod and removes the entry from the apiserver.
- Force deletion of the Pod by the user.

The recommended best practice is to use the first or second approach. If a Node is confirmed to be dead (e.g. permanently disconnected from the network, powered down, etc), then delete the Node object. If the Node is suffering from a network partition, then try to resolve this or wait for it to resolve. When the partition heals, the kubelet will complete the deletion of the Pod and free up its name in the apiserver.

Normally, the system completes the deletion once the Pod is no longer running on a Node, or the Node is deleted by an administrator. You may override this by force deleting the Pod.

Force Deletion

Force deletions **do not** wait for confirmation from the kubelet that the Pod has been terminated. Irrespective of whether a force deletion is successful in killing a Pod, it will immediately free up the name from the apiserver. This would let the StatefulSet controller create a replacement Pod with that same identity; this can lead to the duplication of a still-running Pod, and if said Pod can still communicate with the other members of the StatefulSet, will violate the at most one semantics that StatefulSet is designed to guarantee.

When you force delete a StatefulSet pod, you are asserting that the Pod in question will never again make contact with other Pods in the StatefulSet and its name can be safely freed up for a replacement to be created.

If you want to delete a Pod forcibly using kubectl version >= 1.5, do the following:

```
kubectl delete pods <pod> --grace-period=0 --force
```

If you're using any version of kubectl <= 1.4, you should omit the `--force` option and use:

```
kubectl delete pods <pod> --grace-period=0
```

Always perform force deletion of StatefulSet Pods carefully and with complete knowledge of the risks involved.

What's next

Learn more about debugging a StatefulSet.

Perform Rolling Update Using a Replication Controller

Overview

Note: The preferred way to create a replicated application is to use a Deployment, which in turn uses a ReplicaSet. For more information, see Running a Stateless Application Using a Deployment.

To update a service without an outage, `kubectl` supports what is called 'rolling update', which updates one pod at a time, rather than taking down the entire service at the same time. See the rolling update design document and the example of rolling update for more information.

Note that `kubectl rolling-update` only supports Replication Controllers. However, if you deploy applications with Replication Controllers, consider switching them to Deployments. A Deployment is a higher-level controller that automates rolling updates of applications declaratively, and therefore is recommended. If you still want to keep your Replication Controllers and use `kubectl rolling-update`, keep reading:

A rolling update applies changes to the configuration of pods being managed by a replication controller. The changes can be passed as a new replication controller configuration file; or, if only updating the image, a new container image can be specified directly.

A rolling update works by:

1. Creating a new replication controller with the updated configuration.

2. Increasing/decreasing the replica count on the new and old controllers until the correct number of replicas is reached.
3. Deleting the original replication controller.

Rolling updates are initiated with the `kubectl rolling-update` command:

```
$ kubectl rolling-update NAME \
    ([NEW_NAME] --image=IMAGE | -f FILE)
```

Passing a configuration file

To initiate a rolling update using a configuration file, pass the new file to `kubectl rolling-update`:

```
$ kubectl rolling-update NAME -f FILE
```

The configuration file must:

- Specify a different `metadata.name` value.
- Overwrite at least one common label in its `spec.selector` field.
- Use the same `metadata.namespace`.

Replication controller configuration files are described in Creating Replication Controllers.

Examples

```
// Update pods of frontend-v1 using new replication controller data in frontend-v2
$ kubectl rolling-update frontend-v1 -f frontend-v2.json

// Update pods of frontend-v1 using JSON data passed into stdin.
$ cat frontend-v2.json | kubectl rolling-update frontend-v1 -f -
```

Updating the container image

To update only the container image, pass a new image name and tag with the `--image` flag and (optionally) a new controller name:

```
$ kubectl rolling-update NAME [NEW_NAME] --image=IMAGE:TAG
```

The `--image` flag is only supported for single-container pods. Specifying `--image` with multi-container pods returns an error.

If no `NEW_NAME` is specified, a new replication controller is created with a temporary name. Once the rollout is complete, the old controller is deleted, and the new controller is updated to use the original name.

The update will fail if `IMAGE:TAG` is identical to the current value. For this reason, we recommend the use of versioned tags as opposed to values such as `:latest`. Doing a rolling update from `image:latest` to a new `image:latest` will fail, even if the image at that tag has changed. Moreover, the use of `:latest` is not recommended, see Best Practices for Configuration for more information.

Examples

```
// Update the pods of frontend-v1 to frontend-v2
$ kubectl rolling-update frontend-v1 frontend-v2 --image=image:v2

// Update the pods of frontend, keeping the replication controller name
$ kubectl rolling-update frontend --image=image:v2
```

Required and optional fields

Required fields are:

- `NAME`: The name of the replication controller to update.

as well as either:

- `-f FILE` : A replication controller configuration file, in either JSON or YAML format. The configuration file must specify a new top-level `id` value and include at least one of the existing `spec.selector` key:value pairs. See the [Run Stateless AP Replication Controller]() page for details.

  or:

- `--image IMAGE:TAG` : The name and tag of the image to update to. Must be different than the current image:tag currently specified.

Optional fields are:

- `NEW_NAME` : Only used in conjunction with `--image` (not with `-f FILE`). The name to assign to the new replication controller.
- `--poll-interval DURATION` : The time between polling the controller status after update. Valid units are `ns` (nanoseconds), `us` or `µs` (microseconds), `ms` (milliseconds), `s` (seconds), `m` (minutes), or `h` (hours). Units can be combined (e.g. `1m30s`). The default is `3s` .
- `--timeout DURATION` : The maximum time to wait for the controller to update a pod before exiting. Default is `5m0s` . Valid units are as described for `--poll-interval` above.
- `--update-period DURATION` : The time to wait between updating pods. Default is `1m0s` . Valid units are as described for `--poll-interval` above.

Additional information about the `kubectl rolling-update` command is available from the [`kubectl` reference]().

Walkthrough

Let's say you were running version 1.7.9 of nginx:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

To update to version 1.9.1, you can use `kubectl rolling-update --image` to specify the new image:

```
$ kubectl rolling-update my-nginx --image=nginx:1.9.1
Created my-nginx-ccba8fbd8cc8160970f63f9a2696fc46
```

In another window, you can see that `kubectl` added a `deployment` label to the pods, whose value is a hash of the configuration, to distinguish the new pods from the old:

```
$ kubectl get pods -l app=nginx -L deployment
NAME                                              READY     STATUS    RESTARTS   A
my-nginx-ccba8fbd8cc8160970f63f9a2696fc46-k156z   1/1       Running   0          1
my-nginx-ccba8fbd8cc8160970f63f9a2696fc46-v95yh   1/1       Running   0          3
my-nginx-divi2                                    1/1       Running   0          2
my-nginx-o0ef1                                    1/1       Running   0          2
my-nginx-q6all                                    1/1       Running   0          8
```

`kubectl rolling-update` reports progress as it progresses:

```
Scaling up my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 from 0 to 3, scaling down my-
Scaling my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 up to 1
Scaling my-nginx down to 2
Scaling my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 up to 2
Scaling my-nginx down to 1
Scaling my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 up to 3
Scaling my-nginx down to 0
Update succeeded. Deleting old controller: my-nginx
Renaming my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 to my-nginx
replicationcontroller "my-nginx" rolling updated
```

If you encounter a problem, you can stop the rolling update midway and revert to the previous version using `--rollback`:

```
$ kubectl rolling-update my-nginx --rollback
Setting "my-nginx" replicas to 1
Continuing update with existing controller my-nginx.
Scaling up nginx from 1 to 1, scaling down my-nginx-ccba8fbd8cc8160970f63f9a2696fc
Scaling my-nginx-ccba8fbd8cc8160970f63f9a2696fc46 down to 0
Update succeeded. Deleting my-nginx-ccba8fbd8cc8160970f63f9a2696fc46
replicationcontroller "my-nginx" rolling updated
```

This is one example where the immutability of containers is a huge asset.

If you need to update more than just the image (e.g., command arguments, environment variables), you can create a new replication controller, with a new name and distinguishing label value, such as:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx-v4
spec:
  replicas: 5
  selector:
    app: nginx
    deployment: v4
  template:
    metadata:
      labels:
        app: nginx
        deployment: v4
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.2
        args: ["nginx", "-T"]
        ports:
        - containerPort: 80
```

and roll it out:

```
$ kubectl rolling-update my-nginx -f ./nginx-rc.yaml
Created my-nginx-v4
Scaling up my-nginx-v4 from 0 to 5, scaling down my-nginx from 4 to 0 (keep 4 pods
Scaling my-nginx-v4 up to 1
Scaling my-nginx down to 3
Scaling my-nginx-v4 up to 2
Scaling my-nginx down to 2
Scaling my-nginx-v4 up to 3
Scaling my-nginx down to 1
Scaling my-nginx-v4 up to 4
Scaling my-nginx down to 0
Scaling my-nginx-v4 up to 5
Update succeeded. Deleting old controller: my-nginx
replicationcontroller "my-nginx-v4" rolling updated
```

Troubleshooting

If the `timeout` duration is reached during a rolling update, the operation will fail with some pods belonging to the new replication controller, and some to the original controller.

To continue the update from where it failed, retry using the same command.

To roll back to the original state before the attempted update, append the `--rollback=true` flag to the original command. This will revert all changes.

Horizontal Pod Autoscaling

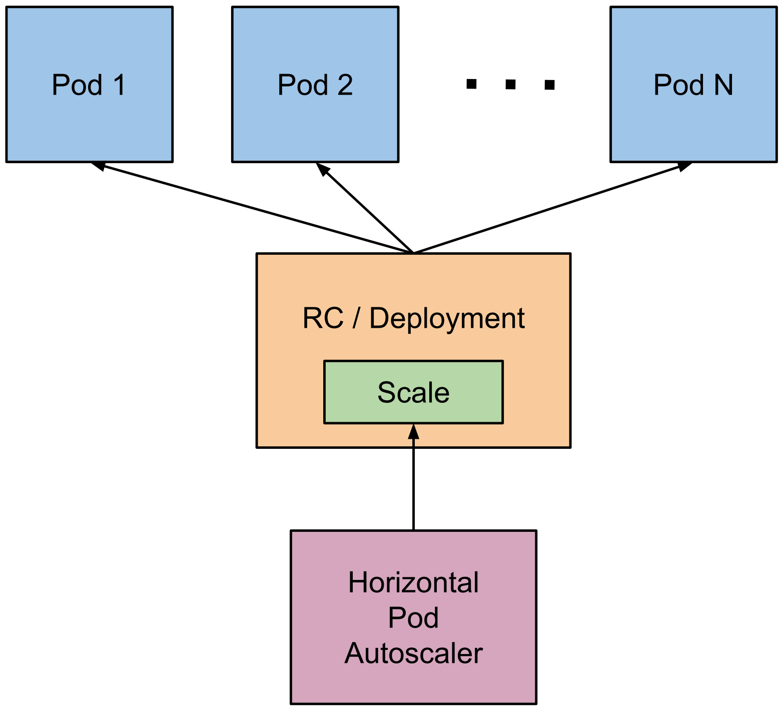
This document describes the current state of Horizontal Pod Autoscaling in Kubernetes.

What is Horizontal Pod Autoscaling?

With Horizontal Pod Autoscaling, Kubernetes automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization (or, with beta support, on some other, application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can't be scaled, for example, DaemonSet.

The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.

How does the Horizontal Pod Autoscaler work?



The Horizontal Pod Autoscaler is implemented as a control loop, with a period controlled by the controller manager's `--horizontal-pod-autoscaler-sync-period` flag (with a default value of 30 seconds).

During each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition. The controller manager obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each pod targeted by the HorizontalPodAutoscaler. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted pods, and produces a ratio used to scale the number of desired replicas.

Please note that if some of the pod's containers do not have the relevant resource request set, CPU utilization for the pod will not be defined and the autoscaler will not take any action for that metric. See the autoscaling algorithm design document for further details about how the autoscaling algorithm works.

- For per-pod custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values, not utilization values.
- For object metrics, a single metric is fetched (which describes the object in question), and compared to the target value, to produce a ratio as above.

The HorizontalPodAutoscaler controller can fetch metrics in two different ways: direct Heapster access, and REST client access.

When using direct Heapster access, the HorizontalPodAutoscaler queries Heapster directly through the API server's service proxy subresource. Heapster needs to be deployed on the cluster and running in the kube-system namespace.

See Support for custom metrics for more details on REST client access.

The autoscaler accesses corresponding replication controller, deployment or replica set by scale sub-resource. Scale is an interface that allows you to dynamically set the number of replicas and examine each of their current states. More details on scale sub-resource can be found here.

API Object

The Horizontal Pod Autoscaler is an API resource in the Kubernetes `autoscaling` API group. The current stable version, which only includes support for CPU autoscaling, can be found in the `autoscaling/v1` API version.

The beta version, which includes support for scaling on memory and custom metrics, can be found in `autoscaling/v2beta1`. The new fields introduced in `autoscaling/v2beta1` are preserved as annotations when working with `autoscaling/v1`.

More details about the API object can be found at HorizontalPodAutoscaler Object.

Support for Horizontal Pod Autoscaler in kubectl

Horizontal Pod Autoscaler, like every API resource, is supported in a standard way by `kubectl`. We can create a new autoscaler using `kubectl create` command. We can list autoscalers by `kubectl get hpa` and get detailed description by `kubectl describe hpa`. Finally, we can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for easy creation of a Horizontal Pod Autoscaler. For instance, executing `kubectl autoscale rc foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for replication controller *foo*, with target CPU utilization set to `80%` and the number of replicas between 2 and 5. The detailed documentation of `kubectl autoscale` can be found here.

Autoscaling during rolling update

Currently in Kubernetes, it is possible to perform a rolling update by managing replication controllers directly, or by using the deployment object, which manages the underlying replication controllers for you. Horizontal Pod Autoscaler only supports the latter approach: the Horizontal Pod Autoscaler is bound to the deployment object, it sets the size for the deployment object, and the deployment is responsible for setting sizes of underlying replication controllers.

Horizontal Pod Autoscaler does not work with rolling update using direct manipulation of replication controllers, i.e. you cannot bind a Horizontal Pod Autoscaler to a replication controller and do rolling update (e.g. using `kubectl rolling-update`). The reason this doesn't work is that when rolling update creates a new replication controller, the Horizontal Pod Autoscaler will not be bound to the new replication controller.

Support for multiple metrics

Kubernetes 1.6 adds support for scaling based on multiple metrics. You can use the `autoscaling/v2beta1` API version to specify multiple metrics for the Horizontal Pod Autoscaler to scale on. Then, the Horizontal Pod Autoscaler controller will evaluate each metric, and propose a new scale based on that metric. The largest of the proposed scales will be used as the new scale.

Support for custom metrics

Note: Kubernetes 1.2 added alpha support for scaling based on application-specific metrics using special annotations. Support for these annotations was removed in Kubernetes 1.6 in favor of the new autoscaling API. While the old method for collecting custom metrics is still available, these metrics will not be available for use by the Horizontal Pod Autoscaler, and the former annotations for specifying which custom metrics to scale on are no longer honored by the Horizontal Pod Autoscaler controller.

Kubernetes 1.6 adds support for making use of custom metrics in the Horizontal Pod Autoscaler. You can add custom metrics for the Horizontal Pod Autoscaler to use in the `autoscaling/v2beta1` API. Kubernetes then queries the new custom metrics API to fetch the values of the appropriate custom metrics.

Requirements

To use custom metrics with your Horizontal Pod Autoscaler, you must set the necessary configurations when deploying your cluster:

- Enable the API aggregation layer if you have not already done so.
- Register your resource metrics API and your custom metrics API with the API aggregation layer. Both of these API servers must be running *on* your cluster.
    - *Resource Metrics API*: You can use Heapster's implementation of the resource metrics API, by running Heapster with its `--api-server` flag set to true.
    - *Custom Metrics API*: This must be provided by a separate component. To get started with boilerplate code, see the kubernetes-incubator/custom-metrics-apiserver and the k8s.io/metrics repositories.
- Set the appropriate flags for kube-controller-manager:
    - `--horizontal-pod-autoscaler-use-rest-clients` should be true.
    - `--kubeconfig <path-to-kubeconfig>` OR `--master <ip-address-of-apiserver>`

      Note that either the `--master` or `--kubeconfig` flag can be used; `--master` will override `--kubeconfig` if both are specified. These flags specify the location of the API aggregation layer, allowing the controller manager to communicate to the API server.

In Kubernetes 1.7, the standard aggregation layer that Kubernetes provides runs in-process with the kube-apiserver, so the target IP address can be found with

```
kubectl get pods --selector k8s-app=kube-apiserver --namespace kube-system
-o jsonpath='{.items[0].status.podIP}'
```

.

Further reading

- Design documentation: Horizontal Pod Autoscaling.
- kubectl autoscale command: kubectl autoscale.
- Usage example of Horizontal Pod Autoscaler.

Horizontal Pod Autoscaling Walkthrough

Horizontal Pod Autoscaling automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization (or, with beta support, on some other, application-provided metrics).

This document walks you through an example of enabling Horizontal Pod Autoscaling for the php-apache server. For more information on how Horizontal Pod Autoscaling behaves, see the Horizontal Pod Autoscaling user guide.

Prerequisites

This example requires a running Kubernetes cluster and kubectl, version 1.2 or later. Heapster monitoring needs to be deployed in the cluster as Horizontal Pod Autoscaler uses it to collect metrics (if you followed getting started on GCE guide, heapster monitoring will be turned-on by default).

To specify multiple resource metrics for a Horizontal Pod Autoscaler, you must have a Kubernetes cluster and kubectl at version 1.6 or later. Furthermore, in order to make use of custom metrics, your cluster must be able to communicate with the API server providing the custom metrics API. See the Horizontal Pod Autoscaling user guide for more details.

Step One: Run & expose php-apache server

To demonstrate Horizontal Pod Autoscaler we will use a custom docker image based on the php-apache image. The Dockerfile can be found here. It defines an index.php page which performs some CPU intensive computations.

First, we will start a deployment running the image and expose it as a service:

```
$ kubectl run php-apache --image=gcr.io/google_containers/hpa-example --requests=c
service "php-apache" created
deployment "php-apache" created
```

Step Two: Create Horizontal Pod Autoscaler

Now that the server is running, we will create the autoscaler using kubectl autoscale. The following command will create a Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment we created in the first step of these instructions. Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50% (since each pod requests 200 milli-cores by kubectl run, this means average CPU usage of 100 milli-cores). See here for more details on the algorithm.

```
$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
deployment "php-apache" autoscaled
```

We may check the current status of autoscaler by running:

```
$ kubectl get hpa
NAME         REFERENCE                     TARGET    MINPODS   MAXPODS   REPLICAS
php-apache   Deployment/php-apache/scale   0% / 50%  1         10        1
```

Please note that the current CPU consumption is 0% as we are not sending any requests to the server (the `CURRENT` column shows the average across all the pods controlled by the corresponding deployment).

Step Three: Increase load

Now, we will see how the autoscaler reacts to increased load. We will start a container, and send an infinite loop of queries to the php-apache service (please run it in a different terminal):

```
$ kubectl run -i --tty load-generator --image=busybox /bin/sh

Hit enter for command prompt

$ while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
```

Within a minute or so, we should see the higher CPU load by executing:

```
$ kubectl get hpa
NAME         REFERENCE                     TARGET      CURRENT   MINPODS   MAXPODS
php-apache   Deployment/php-apache/scale   305% / 50%  305%      1         10
```

Here, CPU consumption has increased to 305% of the request. As a result, the deployment was resized to 7 replicas:

```
$ kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   7         7         7            7           19m
```

Note Sometimes it may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

Step Four: Stop load

We will finish our example by stopping the user load.

In the terminal where we created the container with `busybox` image, terminate the load generation by typing `<Ctrl> + C`.

Then we will verify the result state (after a minute or so):

```
$ kubectl get hpa
NAME         REFERENCE                     TARGET      MINPODS   MAXPODS   REPLIC
php-apache   Deployment/php-apache/scale   0% / 50%    1         10        1

$ kubectl get deployment php-apache
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
php-apache   1         1         1            1           27m
```

Here CPU utilization dropped to 0, and so HPA autoscaled the number of replicas back down to 1.

Note autoscaling the replicas may take a few minutes.

Autoscaling on multiple metrics and custom metrics

You can introduce additional metrics to use when autoscaling the `php-apache` Deployment by making use of the `autoscaling/v2beta1` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2beta1` form:

```
$ kubectl get hpa.v2beta1.autoscaling -o yaml > /tmp/hpa-v2.yaml
```

Open the `/tmp/hpa-v2.yaml` file in an editor, and you should see YAML which looks like this:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 0
```

Notice that the `targetCPUUtilizationPercentage` field has been replaced with an array called `metrics`. The CPU utilization metric is a *resource metric*, since it is represented as a percentage of a resource specified on pod containers. Notice that you can specify other resource metrics besides CPU. By default, the only other supported resource metric is memory. These resources do not change names from cluster to cluster, and should always be available, as long as Heapster is deployed.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value. To do so, use the `targetAverageValue` field instead of the `targetAverageUtilization` field.

There are two other types of metrics, both of which are considered *custom metrics*: pod metrics and object metrics. These metrics may have names which are cluster specific, and require a more advanced cluster monitoring setup.

The first of these alternative metric types is *pod metrics*. These metrics describe pods, and are averaged together across pods and compared with a target value to determine the replica count. They work much like resource metrics, except that they *only* have the `targetAverageValue` field.

Pod metrics are specified using a metric block like this:

```
type: Pods
pods:
  metricName: packets-per-second
  targetAverageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing pods. Note that the metrics are not fetched from the object – they simply describe it. Object metrics do not involve averaging, and look like this:

```
type: Object
object:
  metricName: requests-per-second
  target:
    apiVersion: extensions/v1beta1
    kind: Ingress
    name: main-route
  targetValue: 2k
```

If you provide multiple such metric blocks, the HorizontalPodAutoscaler will consider each metric in turn. The HorizontalPodAutoscaler will calculate proposed replica counts for each metric, and then choose the one with the highest replica count.

For example, if you had your monitoring system collecting metrics about network traffic, you could update the definition above using `kubectl edit` to look like this:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Pods
    pods:
      metricName: packets-per-second
      targetAverageValue: 1k
  - type: Object
    object:
      metricName: requests-per-second
      target:
        apiVersion: extensions/v1beta1
        kind: Ingress
        name: main-route
      targetValue: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 0
```

Then, your HorizontalPodAutoscaler would attempt to ensure that each pod was consuming roughly 50% of its requested CPU, serving 1000 packets per second, and that all pods behind the main-route Ingress were serving a total of 10000 requests per second.

Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2beta1` form of the HorizontalPodAutoscaler, you will be able to see *status conditions* set by Kubernetes on the HorizontalPodAutoscaler. These status conditions indicate whether or not the HorizontalPodAutoscaler is able to scale, and whether or not it is currently restricted in any way.

The conditions appear in the `status.conditions` field. To see the conditions affecting a HorizontalPodAutoscaler, we can use `kubectl describe hpa`:

```
$ kubectl describe hpa cm-test
Name:                           cm-test
Namespace:                      prom
Labels:                         <none>
Annotations:                    <none>
CreationTimestamp:              Fri, 16 Jun 2017 18:09:22 +0000
Reference:                      ReplicationController/cm-test
Metrics:                        ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:                   1
Max replicas:                   4
ReplicationController pods:     1 current / 1 desired
Conditions:
  Type                  Status  Reason                  Message
  ----                  ------  ------                  -------
  AbleToScale           True    ReadyForNewScale        the last scale time was su
  ScalingActive         True    ValidMetricFound        the HPA was able to succes
  ScalingLimited        False   DesiredWithinRange      the desired replica count 
Events:
```

For this HorizontalPodAutoscaler, we can see several conditions in a healthy state. The first, `AbleToScale`, indicates whether or not the HPA is able to fetch and update scales, as well as whether or not any backoff-related conditions would prevent scaling. The second, `ScalingActive`, indicates whether or not the HPA is enabled (i.e. the replica count of the target is not zero) and is able to calculate desired scales. When it is `False`, it generally indicates problems with fetching metrics. Finally, the last condition, `ScalingLimitted`, indicates that the desired scale was capped by the maximum or minimum of the HorizontalPodAutoscaler. This is an indication that you may wish to raise or lower the minimum or maximum replica count constraints on your HorizontalPodAutoscaler.

Appendix: Other possible scenarios

Creating the autoscaler from a .yaml file

Instead of using `kubectl autoscale` command we can use the hpa-php-apache.yaml file, which looks like this:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

We will create the autoscaler by executing the following command:

```
$ kubectl create -f docs/user-guide/horizontal-pod-autoscaling/hpa-php-apache.yaml
horizontalpodautoscaler "php-apache" created
```

Specifying a Disruption Budget for your Application

This page shows how to limit the number of concurrent disruptions that your application experiences, allowing for higher availability while permitting the cluster administrator to manage the clusters nodes.

Before you begin

- You are the owner of an application running on a Kubernetes cluster that requires high availability.
- You should know how to deploy Replicated Stateless Applications and/or Replicated Stateful Applications.
- You should have read about Pod Disruptions.
- You should confirm with your cluster owner or service provider that they respect Pod Disruption Budgets.

Protecting an Application with a PodDisruptionBudget

1. Identify what application you want to protect with a PodDisruptionBudget (PDB).
2. Think about how your application reacts to disruptions.

3. Create a PDB definition as a YAML file.

4. Create the PDB object from the YAML file.

Identify an Application to Protect

The most common use case when you want to protect an application specified by one of the built-in Kubernetes controllers:

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

In this case, make a note of the controller's `.spec.selector`; the same selector goes into the PDBs `.spec.selector`.

You can also use PDBs with pods which are not controlled by one of the above controllers, or arbitrary groups of pods, but there are some restrictions, described in Arbitrary Controllers and Selectors.

Think about how your application reacts to disruptions

Decide how many instances can be down at the same time for a short period due to a voluntary disruption.

- Stateless frontends:
  - Concern: don't reduce serving capacity by more than 10%.
    - Solution: use PDB with minAvailable 90% for example.
- Single-instance Stateful Application:
  - Concern: do not terminate this application without talking to me.

        - Possible Solution 1: Do not use a PDB and tolerate occasional downtime.
        - Possible Solution 2: Set PDB with maxUnavailable=0. Have an understanding (outside of Kubernetes) that the cluster operator needs to consult you before termination. When the cluster operator contacts you, prepare for downtime, and then delete the PDB to indicate readiness for disruption. Recreate afterwards.
- Multiple-instance Stateful application such as Consul, ZooKeeper, or etcd:
    - Concern: Do not reduce number of instances below quorum, otherwise writes fail.
        - Possible Solution 1: set maxUnavailable to 1 (works with varying scale of application).
        - Possible Solution 2: set minAvailable to quorum-size (e.g. 3 when scale is 5). (Allows more disruptions at once).
- Restartable Batch Job:
    - Concern: Job needs to complete in case of voluntary disruption.
        - Possible solution: Do not create a PDB. The Job controller will create a replacement pod.

Specifying a PodDisruptionBudget

A `PodDisruptionBudget` has three fields:

- A label selector `.spec.selector` to specify the set of pods to which it applies. This field is required.
- `.spec.minAvailable` which is a description of the number of pods from that set that must still be available after the eviction, even in the absence of the evicted pod. `minAvailable` can be either an absolute number or a percentage.
- `.spec.maxUnavailable` (available in Kubernetes 1.7 and higher) which is a description of the number of pods from that set that can be unavailable after the eviction. It can be either an absolute number or a percentage.

You can specify only one of `maxUnavailable` and `minAvailable` in a single `PodDisruptionBudget`. `maxUnavailable` can only be used to control the eviction of pods that have an associated controller managing them. In the examples below, “desired replicas” is the `scale` of the controller managing the pods being selected by the `PodDisruptionBudget`.

Example 1: With a `minAvailable` of 5, evictions are be allowed as long as they leave behind 5 or more healthy pods among those selected by the PodDisruptionBudget’s `selector`.

Example 2: With a `minAvailable` of 30%, evictions are allowed as long as at least 30% of the number of desired replicas are healthy.

Example 3: With a `maxUnavailable` of 5, evictions are allowed as long as there are at most 5 unhealthy replicas among the total number of desired replicas.

Example 4: With a `maxUnavailable` of 30%, evictions are allowed as long as no more than 30% of the desired replicas are unhealthy.

In typical usage, a single budget would be used for a collection of pods managed by a controller—for example, the pods in a single ReplicaSet or StatefulSet.

Note: A disruption budget does not truly guarantee that the specified number/percentage of pods will always be up. For example, a node that hosts a pod from the collection may fail when the collection is at the minimum size specified in the budget, thus bringing the number of available pods from the collection below the specified size. The budget can only protect against voluntary evictions, not all causes of unavailability.

A `maxUnavailable` of 0% (or 0) or a `minAvailable` of 100% (or equal to the number of replicas) may block node drains entirely. This is permitted as per the semantics of `PodDisruptionBudget`.

You can find examples of pod disruption budgets defined below. They match pods with the label `app: zookeeper`.

Example PDB Using minAvailable:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

Example PDB Using maxUnavailable (Kubernetes 1.7 or higher):

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

For example, if the above `zk-pdb` object selects the pods of a StatefulSet of size 3, both specifications have the exact same meaning. The use of `maxUnavailable` is recommended as it automatically responds to changes in the number of replicas of the corresponding controller.

Create the PDB object

You can create the PDB object with a command like `kubectl create -f mypdb.yaml`.

You cannot update PDB objects. They must be deleted and re-created.

Check the status of the PDB

Use kubectl to check that your PDB is created.

Assuming you don't actually have pods matching `app: zookeeper` in your namespace, then you'll see something like this:

```
$ kubectl get poddisruptionbudgets
NAME      MIN-AVAILABLE   ALLOWED-DISRUPTIONS   AGE
zk-pdb    2               0                     7s
```

If there are matching pods (say, 3), then you would see something like this:

```
$ kubectl get poddisruptionbudgets
NAME      MIN-AVAILABLE   ALLOWED-DISRUPTIONS   AGE
zk-pdb    2               1                     7s
```

The non-zero value for `ALLOWED-DISRUPTIONS` means that the disruption controller has seen the pods, counted the matching pods, and update the status of the PDB.

You can get more information about the status of a PDB with this command:

```
$ kubectl get poddisruptionbudgets zk-pdb -o yaml
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  creationTimestamp: 2017-08-28T02:38:26Z
  generation: 1
  name: zk-pdb
...
status:
  currentHealthy: 3
  desiredHealthy: 3
  disruptedPods: null
  disruptionsAllowed: 1
  expectedPods: 3
  observedGeneration: 1
```

Arbitrary Controllers and Selectors

You can skip this section if you only use PDBs with the built-in application controllers (Deployment, ReplicationController, ReplicaSet, and StatefulSet), with the PDB selector matching the controller's selector.

You can use a PDB with pods controlled by another type of controller, by an "operator", or bare pods, but with these restrictions:

- only `.spec.minAvailable` can be used, not `.spec.maxUnavailable`.
- only an integer value can be used with `.spec.minAvailable`, not a percentage.

You can use a selector which selects a subset or superset of the pods belonging to a built-in controller. However, when there are multiple PDBs in a namespace, you must be careful not to create PDBs whose selectors overlap.

Parallel Processing using Expansions

Example: Multiple Job Objects from Template Expansion

In this example, we will run multiple Kubernetes Jobs created from a common template. You may want to be familiar with the basic, non-parallel, use of Jobs first.

Basic Template Expansion

First, download the following template of a job to a file called `job.yaml`

job.yaml

job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: busybox
        command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never
```

Unlike a *pod template*, our *job template* is not a Kubernetes API type. It is just a yaml representation of a Job object that has some placeholders that need to be filled in before it can be used. The `$ITEM` syntax is not meaningful to Kubernetes.

In this example, the only processing the container does is to `echo` a string and sleep for a bit. In a real use case, the processing would be some substantial computation, such as rendering a frame of a movie, or processing a range of rows in a database. The "$ITEM" parameter would specify for example, the frame number or the row range.

This Job and its Pod template have a label: `jobgroup=jobexample` . There is nothing special to the system about this label. This label makes it convenient to operate on all the jobs in this group at once. We also put the same label on the pod template so that we can check on all Pods of these Jobs with a single command. After the job is created, the system will add more labels that distinguish one Job's pods from another Job's pods. Note that the label key `jobgroup` is not special to Kubernetes. You can pick your own label scheme.

Next, expand the template into multiple files, one for each item to be processed.

```
# Expand files into a temporary directory
mkdir ./jobs
for i in apple banana cherry
do
  cat job.yaml.txt | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done
```

Check if it worked:

```
$ ls jobs/
job-apple.yaml
job-banana.yaml
job-cherry.yaml
```

Here, we used `sed` to replace the string `$ITEM` with the loop variable. You could use any type of template language (jinja2, erb) or write a program to generate the Job objects.

Next, create all the jobs with one kubectl command:

```
$ kubectl create -f ./jobs
job "process-item-apple" created
job "process-item-banana" created
job "process-item-cherry" created
```

Now, check on the jobs:

```
$ kubectl get jobs -l jobgroup=jobexample
JOB                    CONTAINER(S)   IMAGE(S)   SELECTOR
process-item-apple     c              busybox    app in (jobexample),item in (apple
process-item-banana    c              busybox    app in (jobexample),item in (banan
process-item-cherry    c              busybox    app in (jobexample),item in (cherr
```

Here we use the `-l` option to select all jobs that are part of this group of jobs. (There might be other unrelated jobs in the system that we do not care to see.)

We can check on the pods as well using the same label selector:

```
$ kubectl get pods -l jobgroup=jobexample --show-all
NAME                           READY     STATUS      RESTARTS   AGE
process-item-apple-kixwv       0/1       Completed   0          4m
process-item-banana-wrsf7      0/1       Completed   0          4m
process-item-cherry-dnfu9      0/1       Completed   0          4m
```

There is not a single command to check on the output of all jobs at once, but looping over all the pods is pretty easy:

```
$ for p in $(kubectl get pods -l jobgroup=jobexample --show-all -o name)
do
  kubectl logs $p
done
Processing item apple
Processing item banana
Processing item cherry
```

Multiple Template Parameters

In the first example, each instance of the template had one parameter, and that parameter was also used as a label. However label keys are limited in what characters they can contain.

This slightly more complex example uses the jinja2 template language to generate our objects. We will use a one-line python script to convert the template to a file.

First, copy and paste the following template of a Job object, into a file called `job.yaml.jinja2` :

```
{%- set params = [{ "name": "apple", "url": "http://www.orangepippin.com/apples",
                  { "name": "banana", "url": "https://en.wikipedia.org/wiki/Banana
                  { "name": "raspberry", "url": "https://www.raspberrypi.org/" }]
%}
{%- for p in params %}
{%- set name = p["name"] %}
{%- set url = p["url"] %}
apiVersion: batch/v1
kind: Job
metadata:
  name: jobexample-{{ name }}
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: busybox
        command: ["sh", "-c", "echo Processing URL {{ url }} && sleep 5"]
      restartPolicy: Never
---
{%- endfor %}
```

The above template defines parameters for each job object using a list of python dicts (lines 1-4). Then a for loop emits one job yaml object for each set of parameters (remaining lines). We take advantage of the fact that multiple yaml documents can be concatenated with the `---` separator (second to last line). .) We can pipe the output directly to kubectl to create the objects.

You will need the jinja2 package if you do not already have it: `pip install --user jinja2`. Now, use this one-line python program to expand the template:

```
alias render_template='python -c "from jinja2 import Template; import sys; print(T
```

The output can be saved to a file, like this:

```
cat job.yaml.jinja2 | render_template > jobs.yaml
```

Or sent directly to kubectl, like this:

```
cat job.yaml.jinja2 | render_template | kubectl create -f -
```

Alternatives

If you have a large number of job objects, you may find that:

- Even using labels, managing so many Job objects is cumbersome.
- You exceed resource quota when creating all the Jobs at once, and do not want to wait to create them incrementally.
- You need a way to easily scale the number of pods running concurrently. One reason would be to avoid using too many compute resources. Another would be to limit the number of concurrent requests to a shared resource, such as a database, used by all the pods in the job.
- Very large numbers of jobs created at once overload the Kubernetes apiserver, controller, or scheduler.

In this case, you can consider one of the other job patterns.

Coarse Parallel Processing Using a Work Queue

Example: Job with Work Queue with Pod Per Work Item

In this example, we will run a Kubernetes Job with multiple parallel worker processes. You may want to be familiar with the basic, non-parallel, use of Job first.

In this example, as each pod is created, it picks up one unit of work from a task queue, completes it, deletes it from the queue, and exits.

Here is an overview of the steps in this example:

1. **Start a message queue service.** In this example, we use RabbitMQ, but you could use another one. In practice you would set up a message queue service once and reuse it for many jobs.
2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is just an integer that we will do a lengthy computation on.
3. **Start a Job that works on tasks from the queue**. The Job starts several pods. Each pod takes one task from the message queue, processes it, and repeats until the end of the queue is reached.

Starting a message queue service

This example uses RabbitMQ, but it should be easy to adapt to another AMQP-type message service.

In practice you could set up a message queue service once in a cluster and reuse it for many jobs, as well as for long-running services.

Start RabbitMQ as follows:

```
$ kubectl create -f examples/celery-rabbitmq/rabbitmq-service.yaml
service "rabbitmq-service" created
$ kubectl create -f examples/celery-rabbitmq/rabbitmq-controller.yaml
replicationController "rabbitmq-controller" created
```

We will only use the rabbitmq part from the celery-rabbitmq example.

Testing the message queue service

Now, we can experiment with accessing the message queue. We will create a temporary interactive pod, install some tools on it, and experiment with queues.

First create a temporary interactive Pod.

```
# Create a temporary interactive container
$ kubectl run -i --tty temp --image ubuntu:14.04
Waiting for pod default/temp-loe07 to be running, status is Pending, pod ready: fa
... [ previous line repeats several times .. hit return when it stops ] ...
```

Note that your pod name and command prompt will be different.

Next install the `amqp-tools` so we can work with message queues.

```
# Install some tools
root@temp-loe07:/# apt-get update
.... [ lots of output ] ....
root@temp-loe07:/# apt-get install -y curl ca-certificates amqp-tools python dnsut
.... [ lots of output ] ....
```

Later, we will make a docker image that includes these packages.

Next, we will check that we can discover the rabbitmq service:

```
# Note the rabbitmq-service has a DNS name, provided by Kubernetes:

root@temp-loe07:/# nslookup rabbitmq-service
Server:        10.0.0.10
Address:    10.0.0.10#53

Name:    rabbitmq-service.default.svc.cluster.local
Address: 10.0.147.152

# Your address will vary.
```

If Kube-DNS is not setup correctly, the previous step may not work for you. You can also find the service IP in an env var:

```
# env | grep RABBIT | grep HOST
RABBITMQ_SERVICE_SERVICE_HOST=10.0.147.152
# Your address will vary.
```

Next we will verify we can create a queue, and publish and consume messages.

```
# In the next line, rabbitmq-service is the hostname where the rabbitmq-service
# can be reached.  5672 is the standard port for rabbitmq.

root@temp-loe07:/# export BROKER_URL=amqp://guest:guest@rabbitmq-service:5672
# If you could not resolve "rabbitmq-service" in the previous step,
# then use this command instead:
# root@temp-loe07:/# BROKER_URL=amqp://guest:guest@$RABBITMQ_SERVICE_SERVICE_HOST:

# Now create a queue:

root@temp-loe07:/# /usr/bin/amqp-declare-queue --url=$BROKER_URL -q foo -d
foo

# Publish one message to it:

root@temp-loe07:/# /usr/bin/amqp-publish --url=$BROKER_URL -r foo -p -b Hello

# And get it back.

root@temp-loe07:/# /usr/bin/amqp-consume --url=$BROKER_URL -q foo -c 1 cat && echo
Hello
root@temp-loe07:/#
```

In the last command, the `amqp-consume` tool takes one message (`-c 1`) from the queue, and passes that message to the standard input of an arbitrary command. In this case, the program `cat` is just printing out what it gets on the standard input, and the echo is just to add a carriage return so the example is readable.

Filling the Queue with tasks

Now lets fill the queue with some “tasks”. In our example, our tasks are just strings to be printed.

In a practice, the content of the messages might be:

- names of files to that need to be processed
- extra flags to the program
- ranges of keys in a database table
- configuration parameters to a simulation
- frame numbers of a scene to be rendered

In practice, if there is large data that is needed in a read-only mode by all pods of the Job, you will typically put that in a shared file system like NFS and mount that readonly on all the pods, or the program in the pod will natively read data from a cluster file system like HDFS.

For our example, we will create the queue and fill it using the amqp command line tools. In practice, you might write a program to fill the queue using an amqp client library.

```
$ /usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1  -d
job1
$ for f in apple banana cherry date fig grape lemon melon
do
  /usr/bin/amqp-publish --url=$BROKER_URL -r job1 -p -b $f
done
```

So, we filled the queue with 8 messages.

Create an Image

Now we are ready to create an image that we will run as a job.

We will use the `amqp-consume` utility to read the message from the queue and run our actual program. Here is a very simple example program:

worker.py

```
#!/usr/bin/env python

# Just prints standard out and sleeps for 10 seconds.
import sys
import time
print("Processing " + sys.stdin.lines())
time.sleep(10)
```

Now, build an image. If you are working in the source tree, then change directory to `examples/job/work-queue-1`. Otherwise, make a temporary directory, change to it, download the Dockerfile, and worker.py. In either case, build the image with this command:

```
$ docker build -t job-wq-1 .
```

For the Docker Hub, tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-1 <username>/job-wq-1
docker push <username>/job-wq-1
```

If you are using Google Container Registry, tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```
docker tag job-wq-1 gcr.io/<project>/job-wq-1
gcloud docker -- push gcr.io/<project>/job-wq-1
```

Defining a Job

Here is a job definition. You'll need to make a copy of the Job and edit the image to match the name you used, and call it `./job.yaml`.

job.yaml

job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  completions: 8
  parallelism: 2
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
      - name: c
        image: gcr.io/<project>/job-wq-1
        env:
        - name: BROKER_URL
          value: amqp://guest:guest@rabbitmq-service:5672
        - name: QUEUE
          value: job1
      restartPolicy: OnFailure
```

In this example, each pod works on one item from the queue and then exits. So, the completion count of the Job corresponds to the number of work items done. So we set, `.spec.completions: 8` for the example, since we put 8 items in the queue.

Running the Job

So, now run the Job:

```
kubectl create -f ./job.yaml
```

Now wait a bit, then check on the job.

```
$ kubectl describe jobs/job-wq-1
Name:             job-wq-1
Namespace:        default
Selector:         controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
Labels:           controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
                  job-name=job-wq-1
Annotations:      <none>
Parallelism:      2
Completions:      8
Start Time:       Wed, 06 Sep 2017 16:42:02 +0800
Pods Statuses:    0 Running / 8 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
                job-name=job-wq-1
  Containers:
   c:
    Image:      gcr.io/causal-jigsaw-637/job-wq-1
    Port:
    Environment:
      BROKER_URL:       amqp://guest:guest@rabbitmq-service:5672
      QUEUE:            job1
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen  LastSeen   Count    From    SubobjectPath    Type      Reason
  ─────────  ────────   ─────    ────    ─────────────    ──────    ──────

  27s        27s        1        {job }                   Normal    SuccessfulCrea
  27s        27s        1        {job }                   Normal    SuccessfulCrea
  27s        27s        1        {job }                   Normal    SuccessfulCrea
  27s        27s        1        {job }                   Normal    SuccessfulCrea
  26s        26s        1        {job }                   Normal    SuccessfulCrea
  15s        15s        1        {job }                   Normal    SuccessfulCrea
  14s        14s        1        {job }                   Normal    SuccessfulCrea
  14s        14s        1        {job }                   Normal    SuccessfulCrea
```

All our pods succeeded. Yay.

Alternatives

This approach has the advantage that you do not need to modify your "worker" program to be aware that there is a work queue.

It does require that you run a message queue service. If running a queue service is inconvenient, you may want to consider one of the other job patterns.

This approach creates a pod for every work item. If your work items only take a few seconds, though, creating a Pod for every work item may add a lot of overhead. Consider another example, that executes multiple work items per Pod.

In this example, we used use the `amqp-consume` utility to read the message from the queue and run our actual program. This has the advantage that you do not need to modify your program to be aware of the queue. A different example, shows how to communicate with the work queue using a client library.

Caveats

If the number of completions is set to less than the number of items in the queue, then not all items will be processed.

If the number of completions is set to more than the number of items in the queue, then the Job will not appear to be completed, even though all items in the queue have been processed. It will start additional pods which will block waiting for a message.

There is an unlikely race with this pattern. If the container is killed in between the time that the message is acknowledged by the amqp-consume command and the time that the container exits with success, or if the node crashes before the kubelet is able to post the success of the pod back to the api-server, then the Job will not appear to be complete, even though all items in the queue have been processed.

Fine Parallel Processing Using a Work Queue

Example: Job with Work Queue with Pod Per Work Item

In this example, we will run a Kubernetes Job with multiple parallel worker processes. You may want to be familiar with the basic, non-parallel, use of Job first.

In this example, as each pod is created, it picks up one unit of work from a task queue, completes it, deletes it from the queue, and exits.

Here is an overview of the steps in this example:

1. **Start a storage service to hold the work queue.** In this example, we use Redis to store our work items. In the previous example, we used RabbitMQ. In this example, we use Redis and a custom work-queue client library because AMQP does not provide a good way for clients to detect when a finite-length work queue is empty. In practice you would set up a store such as Redis once and reuse it for the work queues of many jobs, and other things.

2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is just an integer that we will do a lengthy computation on.

3. **Start a Job that works on tasks from the queue**. The Job starts several pods. Each pod takes one task from the message queue, processes it, and repeats until the end of the queue is reached.

Starting Redis

For this example, for simplicity, we will start a single instance of Redis. See the Redis Example for an example of deploying Redis scalably and redundantly.

Start a temporary Pod running Redis and a service so we can find it.

```
$ kubectl create -f docs/tasks/job/fine-parallel-processing-work-queue/redis-pod.y
pod "redis-master" created
$ kubectl create -f docs/tasks/job/fine-parallel-processing-work-queue/redis-servi
service "redis" created
```

If you're not working from the source tree, you could also download `redis-pod.yaml` and `redis-service.yaml` directly.

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are just strings to be printed.

Start a temporary interactive pod for running the Redis CLI.

```
$ kubectl run -i --tty temp --image redis --command "/bin/sh"
Waiting for pod default/redis2-c7h78 to be running, status is Pending, pod ready:
Hit enter for command prompt
```

Now hit enter, start the redis CLI, and create a list with some work items in it.

```
# redis-cli -h redis
redis:6379> rpush job2 "apple"
(integer) 1
redis:6379> rpush job2 "banana"
(integer) 2
redis:6379> rpush job2 "cherry"
(integer) 3
redis:6379> rpush job2 "date"
(integer) 4
redis:6379> rpush job2 "fig"
(integer) 5
redis:6379> rpush job2 "grape"
(integer) 6
redis:6379> rpush job2 "lemon"
(integer) 7
redis:6379> rpush job2 "melon"
(integer) 8
redis:6379> rpush job2 "orange"
(integer) 9
redis:6379> lrange job2 0 -1
1) "apple"
2) "banana"
3) "cherry"
4) "date"
5) "fig"
6) "grape"
7) "lemon"
8) "melon"
9) "orange"
```

So, the list with key `job2` will be our work queue.

Note: if you do not have Kube DNS setup correctly, you may need to change the first step of the above block to `redis-cli -h $REDIS_SERVICE_HOST`.

Create an Image

Now we are ready to create an image that we will run.

We will use a python worker program with a redis client to read the messages from the message queue.

A simple Redis work queue client library is provided, called rediswq.py (Download).

The “worker” program in each Pod of the Job uses the work queue client library to get work. Here it is:

worker.py

```
#!/usr/bin/env python

import time
import rediswq

host="redis"
# Uncomment next two lines if you do not have Kube-DNS working.
# import os
# host = os.getenv("REDIS_SERVICE_HOST")

q = rediswq.RedisWQ(name="job2", host="redis")
print("Worker with sessionID: " +  q.sessionID())
print("Initial queue state: empty=" + str(q.empty()))
while not q.empty():
  item = q.lease(lease_secs=10, block=True, timeout=2)
  if item is not None:
    itemstr = item.decode("utf=8")
    print("Working on " + itemstr)
    time.sleep(10) # Put your actual work here instead of sleep.
    q.complete(item)
  else:
    print("Waiting for work")
print("Queue empty, exiting")
```

If you are working from the source tree, change directory to the `docs/tasks/job/fine-parallel-processing-work-queue/` directory. Otherwise, download `worker.py`, `rediswq.py`, and `Dockerfile` using above links. Then build the image:

```
docker build -t job-wq-2 .
```

Push the image

For the Docker Hub, tag your app image with your username and push to the Hub with the below commands. Replace `<username>` with your Hub username.

```
docker tag job-wq-2 <username>/job-wq-2
docker push <username>/job-wq-2
```

You need to push to a public repository or configure your cluster to be able to access your private repository.

If you are using Google Container Registry, tag your app image with your project ID, and push to GCR. Replace `<project>` with your project ID.

```
docker tag job-wq-2 gcr.io/<project>/job-wq-2
gcloud docker -- push gcr.io/<project>/job-wq-2
```

Defining a Job

Here is the job definition:

job.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-2
spec:
  parallelism: 2
  template:
    metadata:
      name: job-wq-2
    spec:
      containers:
      - name: c
        image: gcr.io/myproject/job-wq-2
      restartPolicy: OnFailure
```

Be sure to edit the job template to change `gcr.io/myproject` to your own path.

In this example, each pod works on several items from the queue and then exits when there are no more items. Since the workers themselves detect when the workqueue is empty, and the Job controller does not know about the workqueue, it relies on the workers to signal when they are done working. The workers signal that the queue is empty by exiting with success. So, as soon as any worker exits with success, the controller knows the work is done, and the Pods will exit soon. So, we set the completion count of the Job to 1. The job controller will wait for the other pods to complete too.

Running the Job

So, now run the Job:

```
kubectl create -f ./job.yaml
```

Now wait a bit, then check on the job.

```
$ kubectl describe jobs/job-wq-2
Name:             job-wq-2
Namespace:        default
Selector:         controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
Labels:           controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
                  job-name=job-wq-2
Annotations:      <none>
Parallelism:      2
Completions:      <unset>
Start Time:       Mon, 11 Jan 2016 17:07:59 -0800
Pods Statuses:    1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
                job-name=job-wq-2
  Containers:
   c:
    Image:              gcr.io/exampleproject/job-wq-2
    Port:
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen    LastSeen    Count    From            SubobjectPath    Type        R
  ---------    --------    -----    ----            -------------    --------    -
  33s          33s         1        {job-controller }                Normal      S


$ kubectl logs pods/job-wq-2-7r7b2
Worker with sessionID: bbd72d0a-9e5c-4dd6-abf6-416cc267991f
Initial queue state: empty=False
Working on banana
Working on date
Working on lemon
```

As you can see, one of our pods worked on several work units.

Alternatives

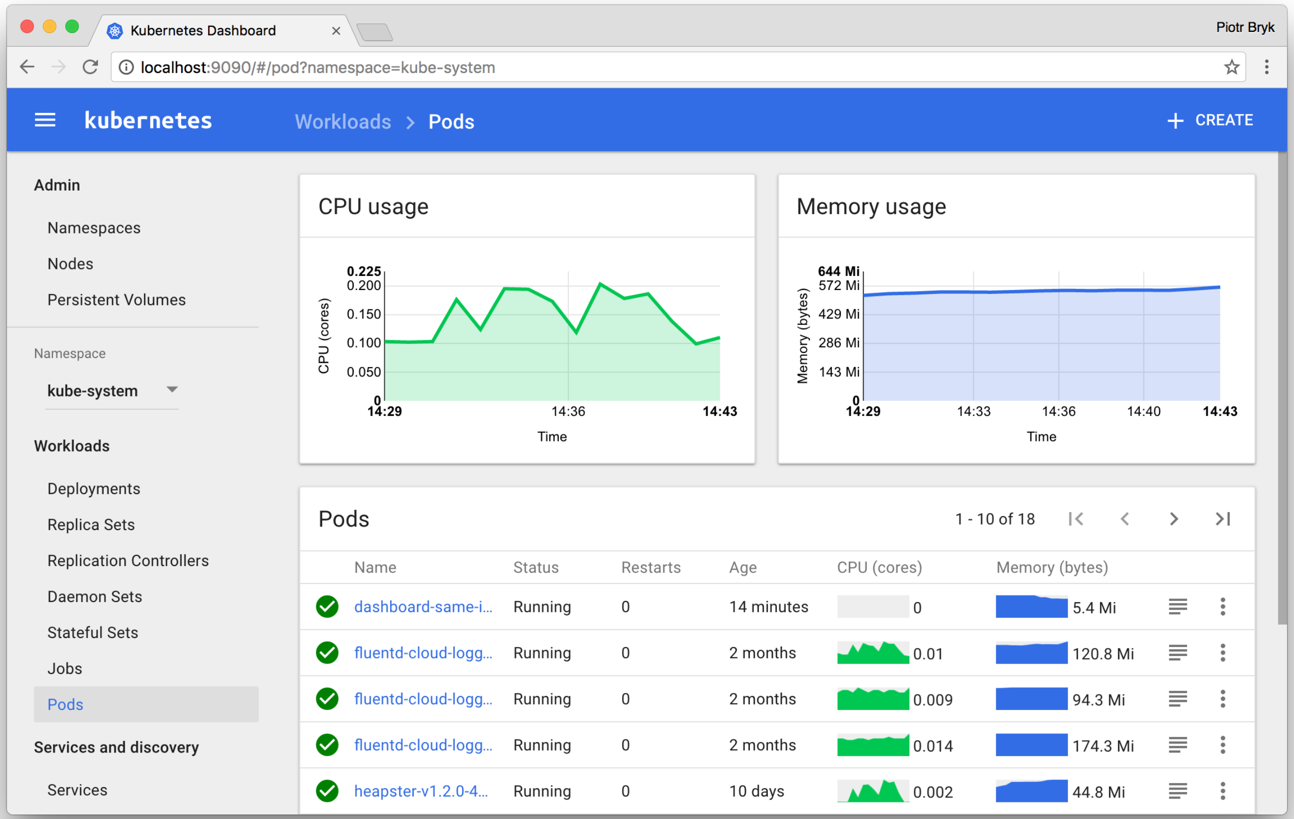
If running a queue service or modifying your containers to use a work queue is inconvenient, you may want to consider one of the other job patterns.

If you have a continuous stream of background processing work to run, then consider running your background workers with a `replicationController` instead, and consider running a background processing library such as https://github.com/resque/resque.

Web UI (Dashboard)

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster itself along with its attendant resources. You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). For example, you can scale a Deployment, initiate a rolling update, restart a pod or deploy new applications using a deploy wizard.

Dashboard also provides information on the state of Kubernetes resources in your cluster, and on any errors that may have occurred.



	Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓	dashboard-same-i...	Running	0	14 minutes	0	5.4 Mi
✓	fluentd-cloud-logg...	Running	0	2 months	0.01	120.8 Mi
✓	fluentd-cloud-logg...	Running	0	2 months	0.009	94.3 Mi
✓	fluentd-cloud-logg...	Running	0	2 months	0.014	174.3 Mi
✓	heapster-v1.2.0-4...	Running	0	10 days	0.002	44.8 Mi

Deploying the Dashboard UI

The Dashboard UI is not deployed by default. To deploy it, run the following command:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/sr
```

Accessing the Dashboard UI

There are multiple ways you can access the Dashboard UI; either by using the kubectl command-line interface, or by accessing the Kubernetes master apiserver using your web browser.

Command line proxy

You can access Dashboard using the kubectl command-line tool by running the following command:

```
$ kubectl proxy
```

Kubectl will handle authentication with apiserver and make Dashboard available at http://localhost:8001/ui.

The UI can *only* be accessed from the machine where the command is executed. See `kubectl proxy --help` for more options.

Master server

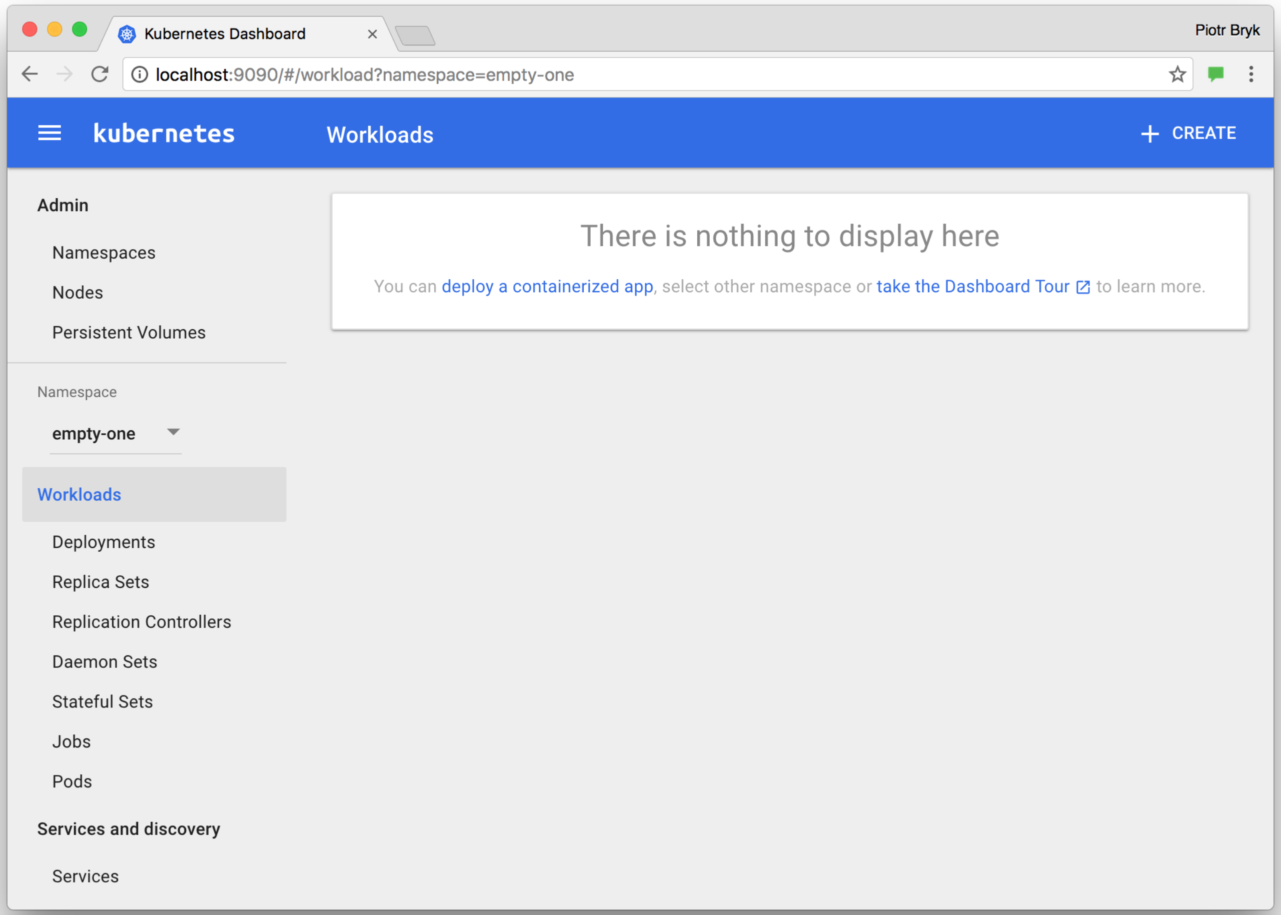
You may access the UI directly via the Kubernetes master apiserver. Open a browser and navigate to `https://<kubernetes-master>/ui`, where `<kubernetes-master>` is IP address or domain name of the Kubernetes master.

Please note, this works only if the apiserver is set up to allow authentication with username and password. This is not currently the case with some setup tools (e.g., `kubeadm`). Refer to the authentication admin documentation for information on how to configure authentication manually.

If the username and password are configured but unknown to you, then use `kubectl config view` to find it.

Welcome view

When you access Dashboard on an empty cluster, you'll see the welcome page. This page contains a link to this document as well as a button to deploy your first application. In addition, you can view which system applications are running by default in the `kube-system` namespace of your cluster, for example the Dashboard itself.



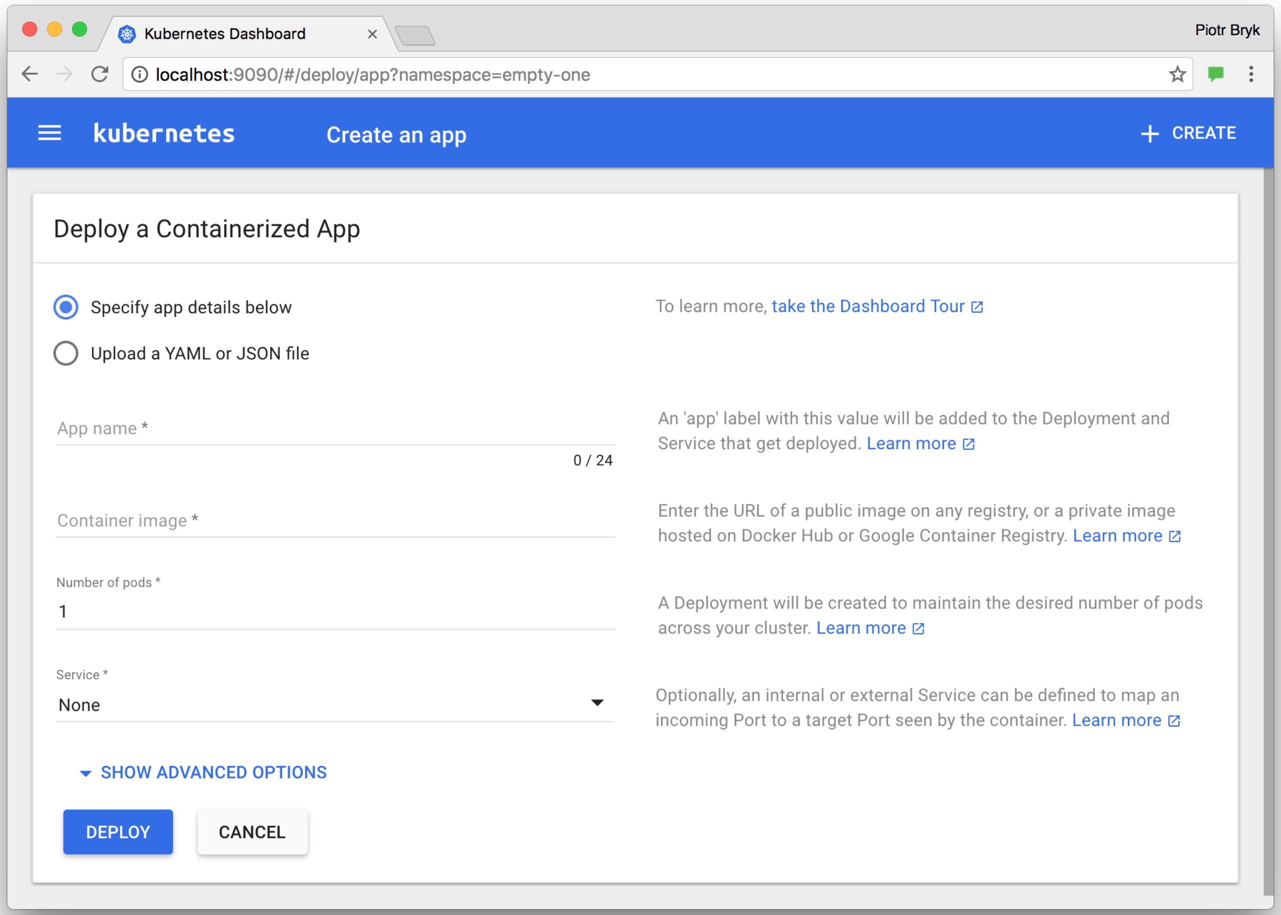
There is nothing to display here

You can deploy a containerized app, select other namespace or take the Dashboard Tour to learn more.

Deploying containerized applications

Dashboard lets you create and deploy a containerized application as a Deployment and optional Service with a simple wizard. You can either manually specify application details, or upload a YAML or JSON file containing application configuration.

To access the deploy wizard from the Welcome page, click the respective button. To access the wizard at a later point in time, click the **CREATE** button in the upper right corner of any page.



Deploy a Containerized App

Specify app details below [✓]

Upload a YAML or JSON file

To learn more, take the Dashboard Tour

App name *

0 / 24

An 'app' label with this value will be added to the Deployment and Service that get deployed. Learn more

Container image *

Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. Learn more

Number of pods *

1

A Deployment will be created to maintain the desired number of pods across your cluster. Learn more

Service *

None

Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. Learn more

SHOW ADVANCED OPTIONS

DEPLOY CANCEL

Specifying application details

The deploy wizard expects that you provide the following information:

- **App name** (mandatory): Name for your application. A label with the name will be added to the Deployment and Service, if any, that will be deployed.

  The application name must be unique within the selected Kubernetes namespace. It must start with a lowercase character, and end with a lowercase character or a number, and contain only lowercase letters, numbers and dashes (-). It is limited to 24 characters. Leading and trailing spaces are ignored.

- **Container image** (mandatory): The URL of a public Docker container image on any registry, or a private image (commonly hosted on the Google Container Registry or Docker Hub). The container image specification must end with a colon.

- **Number of pods** (mandatory): The target number of Pods you want your application to be deployed in. The value must be a positive integer.

  A Deployment will be created to maintain the desired number of Pods across your cluster.

- **Service** (optional): For some parts of your application (e.g. frontends) you may want to expose a Service onto an external, maybe public IP address outside of your cluster (external Service). For external Services, you may need to open up one or more ports to do so. Find more details here.

  Other Services that are only visible from inside the cluster are called internal Services.

  Irrespective of the Service type, if you choose to create a Service and your container listens on a port (incoming), you need to specify two ports. The Service will be created mapping the port (incoming) to the target port seen by the container. This Service will route to your deployed Pods. Supported protocols are TCP and UDP. The internal DNS name for this Service will be the value you specified as application name above.

If needed, you can expand the **Advanced options** section where you can specify more settings:

- **Description**: The text you enter here will be added as an annotation to the Deployment and displayed in the application's details.

- **Labels**: Default labels to be used for your application are application name and version. You can specify additional labels to be applied to the Deployment, Service (if any), and Pods, such as release, environment, tier, partition, and release track.

  Example:

  ```
  release=1.0
  tier=frontend
  environment=pod
  track=stable
  ```

- **Namespace**: Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces. They let you partition resources into logically named groups.

  Dashboard offers all available namespaces in a dropdown list, and allows you to create a new namespace. The namespace name may contain a maximum of 63 alphanumeric characters and dashes (-) but can not contain capital letters. Namespace names should not consist of only numbers. If the name is set as a number, such as 10, the pod will be put in the default namespace.

In case the creation of the namespace is successful, it is selected by default. If the creation fails, the first namespace is selected.

- **Image Pull Secret**: In case the specified Docker container image is private, it may require pull secret credentials.

  Dashboard offers all available secrets in a dropdown list, and allows you to create a new secret. The secret name must follow the DNS domain name syntax, e.g. `new.image-pull.secret`. The content of a secret must be base64-encoded and specified in a `.dockercfg` file. The secret name may consist of a maximum of 253 characters.

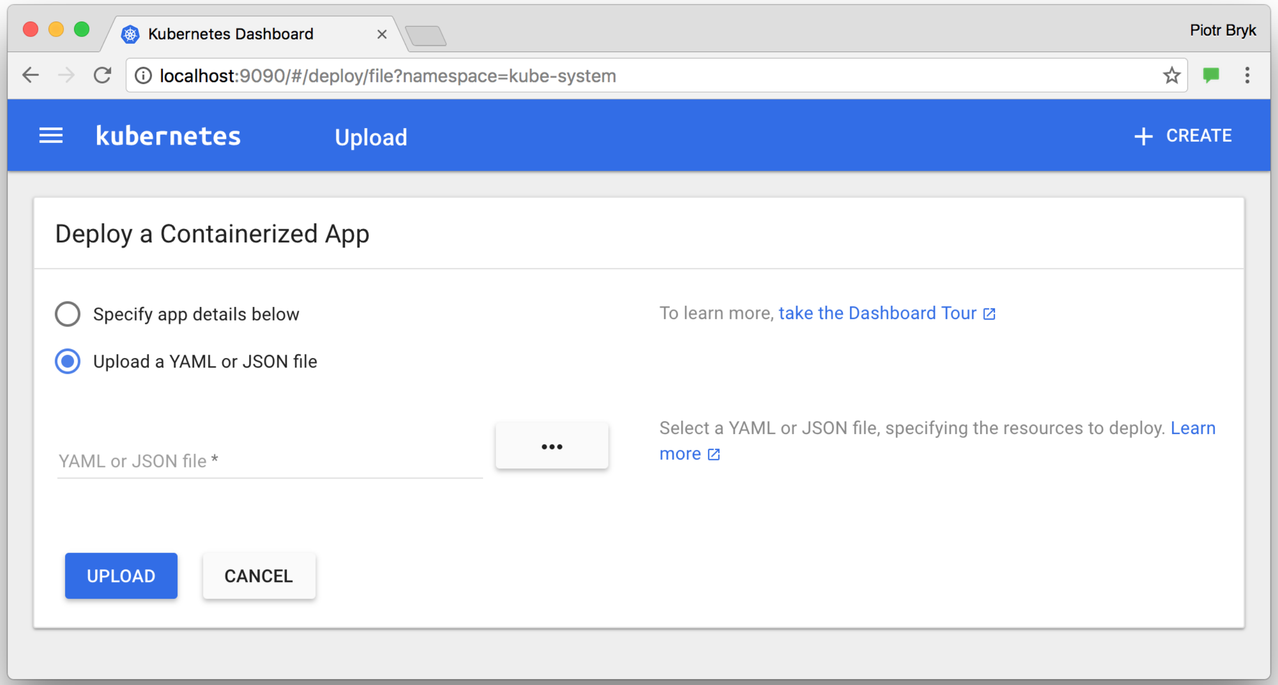
  In case the creation of the image pull secret is successful, it is selected by default. If the creation fails, no secret is applied.

- **CPU requirement (cores)** and **Memory requirement (MiB)**: You can specify the minimum resource limits for the container. By default, Pods run with unbounded CPU and memory limits.
- **Run command** and **Run command arguments**: By default, your containers run the specified Docker image's default entrypoint command. You can use the command options and arguments to override the default.
- **Run as privileged**: This setting determines whether processes in privileged containers are equivalent to processes running as root on the host. Privileged containers can make use of capabilities like manipulating the network stack and accessing devices.
- **Environment variables**: Kubernetes exposes Services through environment variables. You can compose environment variable or pass arguments to your commands using the values of environment variables. They can be used in applications to find a Service. Values can reference other variables using the `$(VAR_NAME)` syntax.

Uploading a YAML or JSON file

Kubernetes supports declarative configuration. In this style, all configuration is stored in YAML or JSON configuration files using the Kubernetes API resource schemas.

As an alternative to specifying application details in the deploy wizard, you can define your application in YAML or JSON files, and upload the files using Dashboard:



Deploy a Containerized App

Specify app details below

Upload a YAML or JSON file

To learn more, take the Dashboard Tour

YAML or JSON file *

Select a YAML or JSON file, specifying the resources to deploy. Learn more

Using Dashboard

Following sections describe views of the Kubernetes Dashboard UI; what they provide and how can they be used.

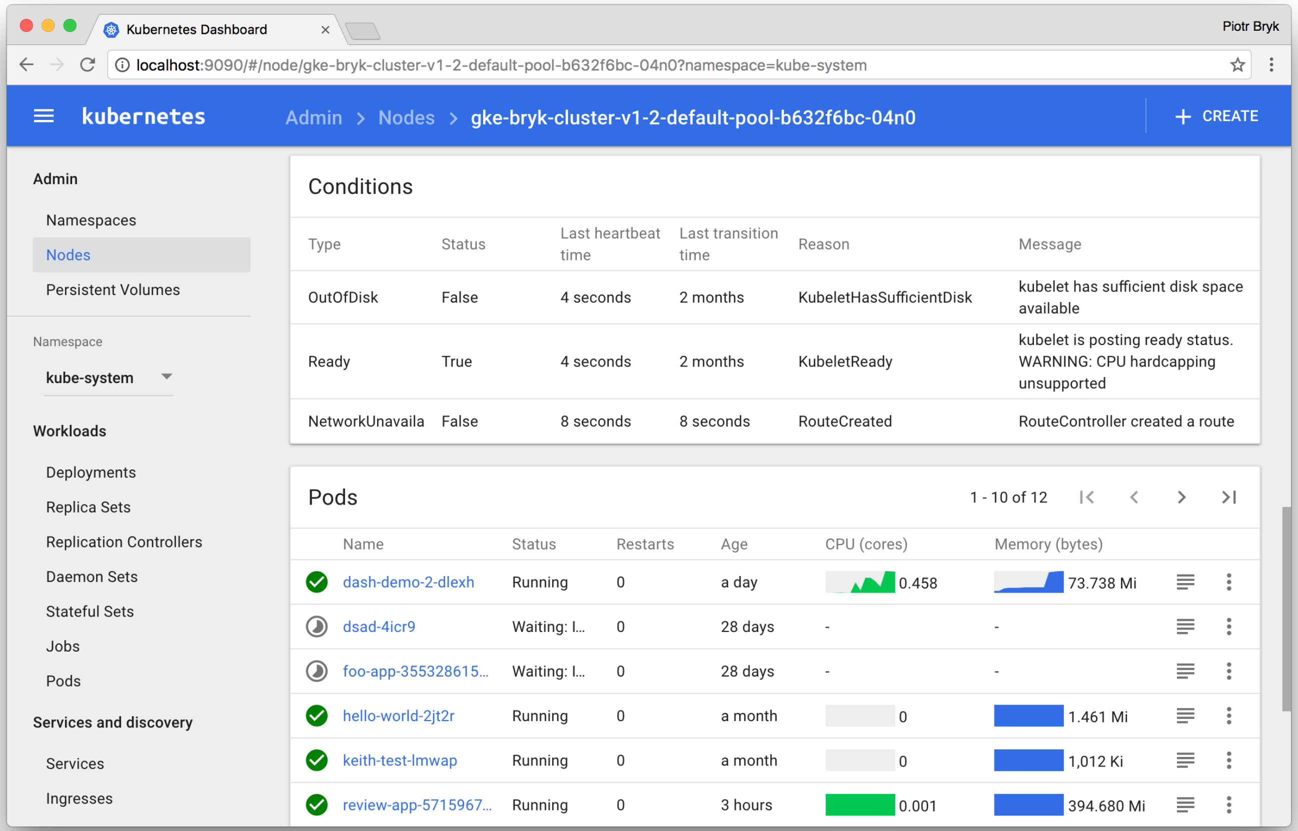
Navigation

When there are Kubernetes objects defined in the cluster, Dashboard shows them in the initial view. By default only objects from the *default* namespace are shown and this can be changed using the namespace selector located in the navigation menu.

Dashboard shows most Kubernetes object kinds and groups them in a few menu categories.

Admin

View for cluster and namespace administrators. It lists Nodes, Namespaces and Persistent Volumes and has detail views for them. Node list view contains CPU and memory usage metrics aggregated across all Nodes. The details view shows the metrics for a Node, its specification, status, allocated resources, events and pods running on the node.



Conditions

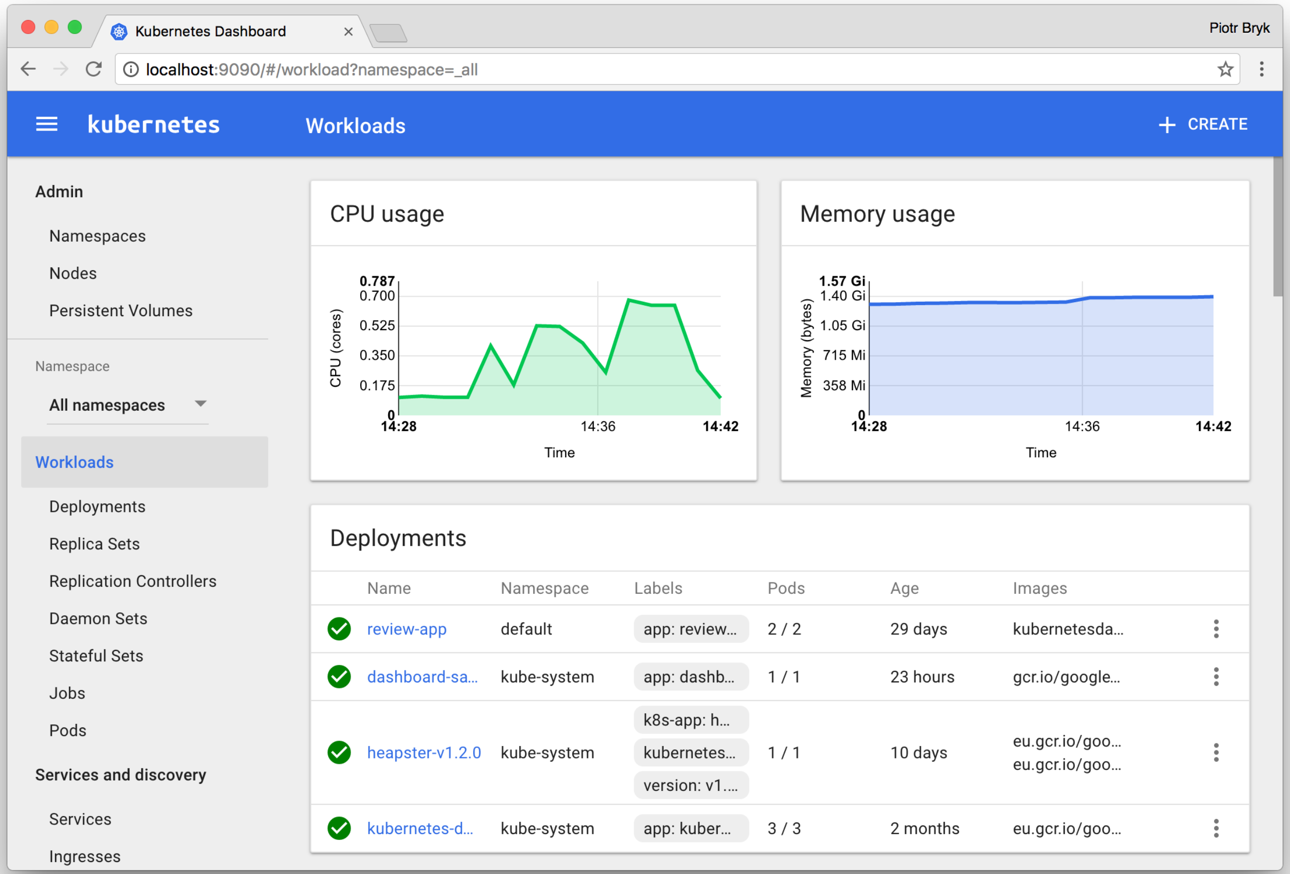
Type	Status	Last heartbeat time	Last transition time	Reason	Message
OutOfDisk	False	4 seconds	2 months	KubeletHasSufficientDisk	kubelet has sufficient disk space available
Ready	True	4 seconds	2 months	KubeletReady	kubelet is posting ready status. WARNING: CPU hardcapping unsupported
NetworkUnavaila	False	8 seconds	8 seconds	RouteCreated	RouteController created a route

Pods 1 - 10 of 12

	Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓	dash-demo-2-dlexh	Running	0	a day	0.458	73.738 Mi
◔	dsad-4icr9	Waiting: I...	0	28 days	-	-
◔	foo-app-355328615...	Waiting: I...	0	28 days	-	-
✓	hello-world-2jt2r	Running	0	a month	0	1.461 Mi
✓	keith-test-lmwap	Running	0	a month	0	1,012 Ki
✓	review-app-5715967...	Running	0	3 hours	0.001	394.680 Mi

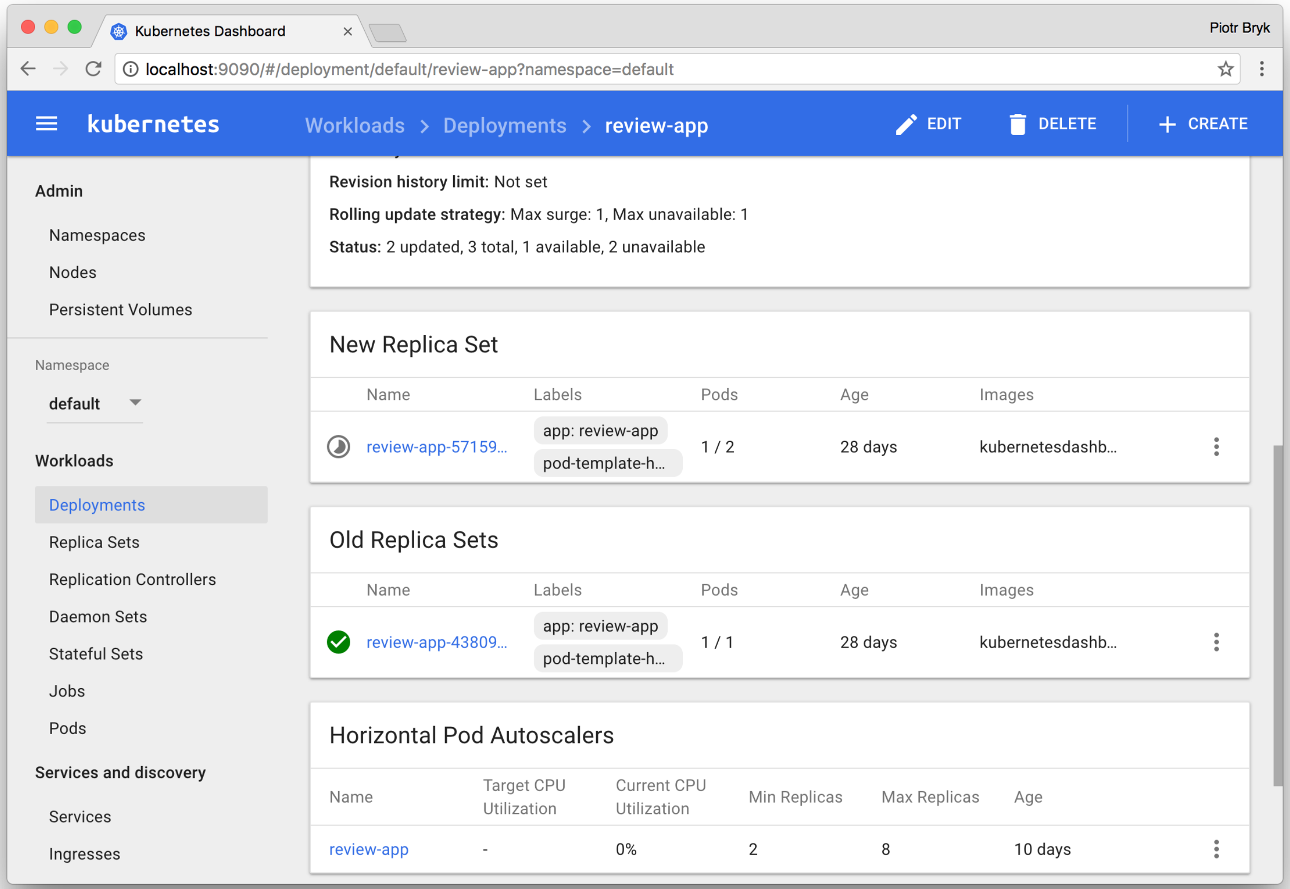
Workloads

Entry point view that shows all applications running in the selected namespace. The view lists applications by workload kind (e.g., Deployments, Replica Sets, Stateful Sets, etc.) and each workload kind can be viewed separately. The lists summarize actionable information about the workloads, such as the number of ready pods for a Replica Set or current memory usage for a Pod.



Name	Namespace	Labels	Pods	Age	Images
review-app	default	app: review...	2 / 2	29 days	kubernetesda...
dashboard-sa...	kube-system	app: dashb...	1 / 1	23 hours	gcr.io/google...
heapster-v1.2.0	kube-system	k8s-app: h... kubernetes... version: v1....	1 / 1	10 days	eu.gcr.io/goo... eu.gcr.io/goo...
kubernetes-d...	kube-system	app: kuber...	3 / 3	2 months	eu.gcr.io/goo...

Detail views for workloads show status and specification information and surface relationships between objects. For example, Pods that Replica Set is controlling or New Replica Sets and Horizontal Pod Autoscalers for Deployments.



Revision history limit: Not set

Rolling update strategy: Max surge: 1, Max unavailable: 1

Status: 2 updated, 3 total, 1 available, 2 unavailable

New Replica Set

Name	Labels	Pods	Age	Images
review-app-57159...	app: review-app pod-template-h...	1 / 2	28 days	kubernetesdashb...

Old Replica Sets

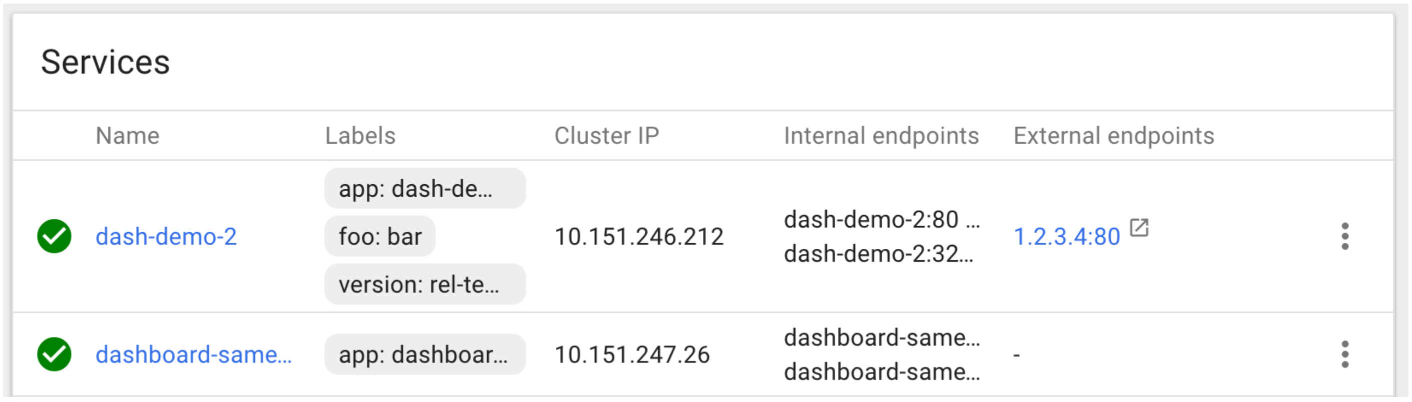
Name	Labels	Pods	Age	Images
✓ review-app-43809...	app: review-app pod-template-h...	1 / 1	28 days	kubernetesdashb...

Horizontal Pod Autoscalers

Name	Target CPU Utilization	Current CPU Utilization	Min Replicas	Max Replicas	Age
review-app	-	0%	2	8	10 days

Services and discovery

Services and discovery view shows Kubernetes resources that allow for exposing services to external world and discovering them within a cluster. For that reason, Service and Ingress views show Pods targeted by them, internal endpoints for cluster connections and external endpoints for external users.



Services

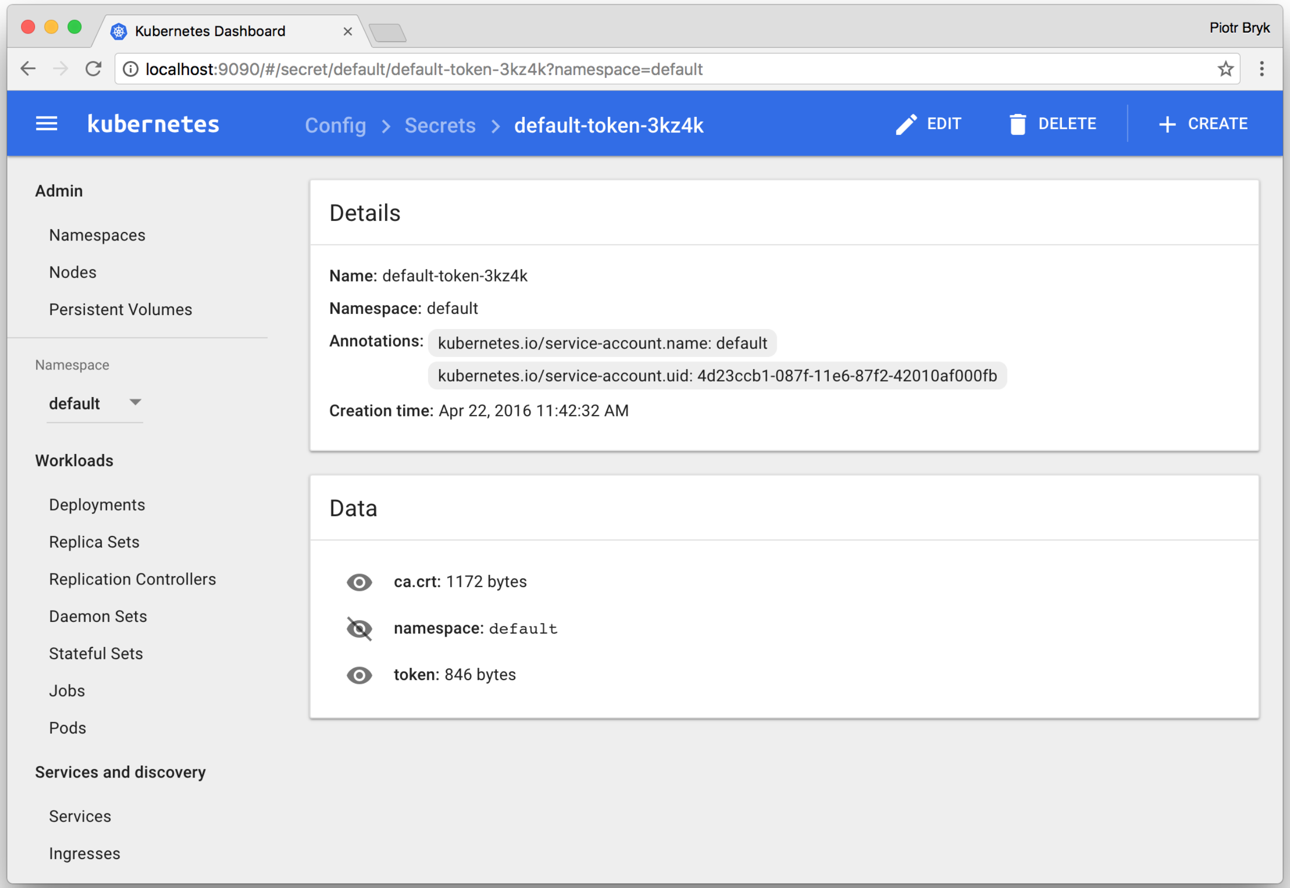
Name	Labels	Cluster IP	Internal endpoints	External endpoints
✓ dash-demo-2	app: dash-de... foo: bar version: rel-te...	10.151.246.212	dash-demo-2:80 ... dash-demo-2:32...	1.2.3.4:80
✓ dashboard-same...	app: dashboar...	10.151.247.26	dashboard-same... dashboard-same...	-

Storage

Storage view shows Persistent Volume Claim resources which are used by applications for storing data.

Config

Config view show all Kubernetes resources that are used for live configuration of applications running in clusters. This is now Config Maps and Secrets. The view allows for editing and managing config objects and displays secrets hidden by default.


Kubernetes Dashboard

localhost:9090/#/secret/default/default-token-3kz4k?namespace=default

Details

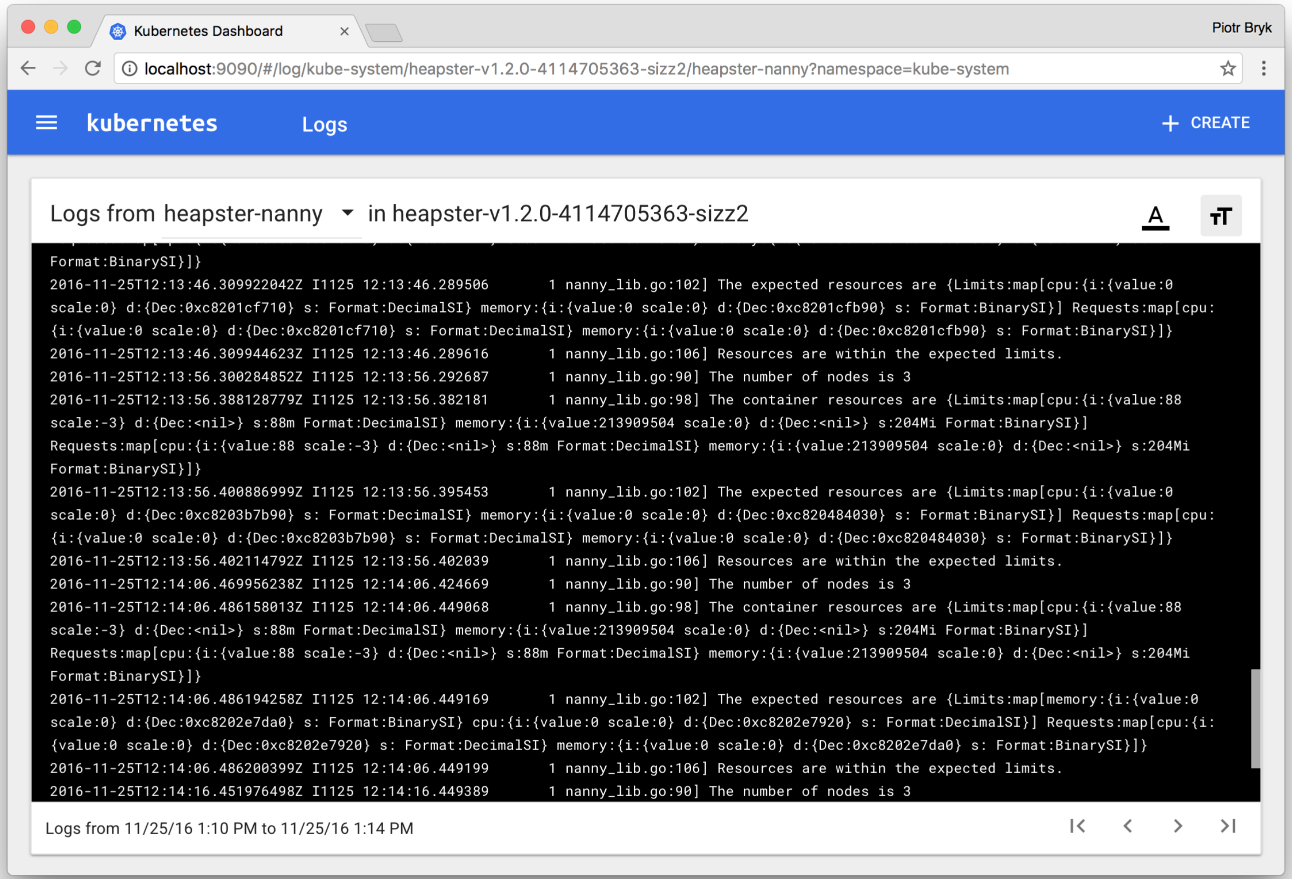
Name: default-token-3kz4k
Namespace: default
Annotations: kubernetes.io/service-account.name: default
kubernetes.io/service-account.uid: 4d23ccb1-087f-11e6-87f2-42010af000fb
Creation time: Apr 22, 2016 11:42:32 AM

Data

ca.crt: 1172 bytes
namespace: `default`
token: 846 bytes

Logs viewer

Pod lists and detail pages link to logs viewer that is built into Dashboard. The viewer allows for drilling down logs from containers belonging to a single Pod.


Kubernetes Dashboard
Piotr Bryk
localhost:9090/#/log/kube-system/heapster-v1.2.0-4114705363-sizz2/heapster-nanny?namespace=kube-system
kubernetes
Logs
CREATE

Logs from heapster-nanny in heapster-v1.2.0-4114705363-sizz2

```
Format:BinarySI}]}
2016-11-25T12:13:46.309922042Z I1125 12:13:46.289506       1 nanny_lib.go:102] The expected resources are {Limits:map[cpu:{i:{value:0
scale:0} d:{Dec:0xc8201cf710} s: Format:DecimalSI} memory:{i:{value:0 scale:0} d:{Dec:0xc8201cfb90} s: Format:BinarySI}] Requests:map[cpu:
{i:{value:0 scale:0} d:{Dec:0xc8201cf710} s: Format:DecimalSI} memory:{i:{value:0 scale:0} d:{Dec:0xc8201cfb90} s: Format:BinarySI}]}
2016-11-25T12:13:46.309944623Z I1125 12:13:46.289616       1 nanny_lib.go:106] Resources are within the expected limits.
2016-11-25T12:13:56.300284852Z I1125 12:13:56.292687       1 nanny_lib.go:90] The number of nodes is 3
2016-11-25T12:13:56.388128779Z I1125 12:13:56.382181       1 nanny_lib.go:98] The container resources are {Limits:map[cpu:{i:{value:88
scale:-3} d:{Dec:<nil>} s:88m Format:DecimalSI} memory:{i:{value:213909504 scale:0} d:{Dec:<nil>} s:204Mi Format:BinarySI}]
Requests:map[cpu:{i:{value:88 scale:-3} d:{Dec:<nil>} s:88m Format:DecimalSI} memory:{i:{value:213909504 scale:0} d:{Dec:<nil>} s:204Mi
Format:BinarySI}]}
2016-11-25T12:13:56.400886999Z I1125 12:13:56.395453       1 nanny_lib.go:102] The expected resources are {Limits:map[cpu:{i:{value:0
scale:0} d:{Dec:0xc8203b7b90} s: Format:DecimalSI} memory:{i:{value:0 scale:0} d:{Dec:0xc820484030} s: Format:BinarySI}] Requests:map[cpu:
{i:{value:0 scale:0} d:{Dec:0xc8203b7b90} s: Format:DecimalSI} memory:{i:{value:0 scale:0} d:{Dec:0xc820484030} s: Format:BinarySI}]}
2016-11-25T12:13:56.402114792Z I1125 12:13:56.402039       1 nanny_lib.go:106] Resources are within the expected limits.
2016-11-25T12:14:06.469956238Z I1125 12:14:06.424669       1 nanny_lib.go:90] The number of nodes is 3
2016-11-25T12:14:06.486158013Z I1125 12:14:06.449068       1 nanny_lib.go:98] The container resources are {Limits:map[cpu:{i:{value:88
scale:-3} d:{Dec:<nil>} s:88m Format:DecimalSI} memory:{i:{value:213909504 scale:0} d:{Dec:<nil>} s:204Mi Format:BinarySI}]
Requests:map[cpu:{i:{value:88 scale:-3} d:{Dec:<nil>} s:88m Format:DecimalSI} memory:{i:{value:213909504 scale:0} d:{Dec:<nil>} s:204Mi
Format:BinarySI}]}
2016-11-25T12:14:06.486194258Z I1125 12:14:06.449169       1 nanny_lib.go:102] The expected resources are {Limits:map[memory:{i:{value:0
scale:0} d:{Dec:0xc8202e7da0} s: Format:BinarySI} cpu:{i:{value:0 scale:0} d:{Dec:0xc8202e7920} s: Format:DecimalSI}] Requests:map[cpu:{i:
{value:0 scale:0} d:{Dec:0xc8202e7920} s: Format:DecimalSI} memory:{i:{value:0 scale:0} d:{Dec:0xc8202e7da0} s: Format:BinarySI}]}
2016-11-25T12:14:06.486200399Z I1125 12:14:06.449199       1 nanny_lib.go:106] Resources are within the expected limits.
2016-11-25T12:14:16.451976498Z I1125 12:14:16.449389       1 nanny_lib.go:90] The number of nodes is 3
```

Logs from 11/25/16 1:10 PM to 11/25/16 1:14 PM

More information

For more information, see the Kubernetes Dashboard project page.

Accessing Clusters

Accessing the cluster API

Accessing for the first time with kubectl

When accessing the Kubernetes API for the first time, we suggest using the Kubernetes CLI, `kubectl`.

To access a cluster, you need to know the location of the cluster and have credentials to access it. Typically, this is automatically set-up when you work through a Getting started guide, or someone else setup the cluster and provided you with credentials and a location.

Check the location and credentials that kubectl knows about with this command:

```
$ kubectl config view
```

Many of the examples provide an introduction to using kubectl and complete documentation is found in the kubectl manual.

Directly accessing the REST API

Kubectl handles locating and authenticating to the apiserver. If you want to directly access the REST API with an http client like curl or wget, or a browser, there are several ways to locate and authenticate:

- Run kubectl in proxy mode.
    - Recommended approach.
    - Uses stored apiserver location.
    - Verifies identity of apiserver using self-signed cert. No MITM possible.
    - Authenticates to apiserver.
    - In future, may do intelligent client-side load-balancing and failover.
- Provide the location and credentials directly to the http client.
    - Alternate approach.
    - Works with some types of client code that are confused by using a proxy.
    - Need to import a root cert into your browser to protect against MITM.

Using kubectl proxy

The following command runs kubectl in a mode where it acts as a reverse proxy. It handles locating the apiserver and authenticating. Run it like this:

```
$ kubectl proxy --port=8080 &
```

See kubectl proxy for more details.

Then you can explore the API with curl, wget, or a browser, like so:

```
$ curl http://localhost:8080/api/
{
  "versions": [
    "v1"
  ]
}
```

Without kubectl proxy (before v1.3.x)

It is possible to avoid using kubectl proxy by passing an authentication token directly to the apiserver, like this:

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl config view | grep token | cut -f 2 -d ":" | tr -d " ")
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "versions": [
    "v1"
  ]
}
```

Without kubectl proxy (post v1.3.x)

In Kubernetes version 1.3 or later, `kubectl config view` no longer displays the token. Use `kubectl describe secret...` to get the token for the default service account, like this:

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

The above examples use the `--insecure` flag. This leaves it subject to MITM attacks. When kubectl accesses the cluster it uses a stored root certificate and client certificates to access the server. (These are installed in the `~/.kube` directory). Since cluster certificates are typically self-signed, it may take special configuration to get your http client to use root certificate.

On some clusters, the apiserver does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. Configuring Access to the API describes how a cluster admin can configure this. Such approaches may conflict with future high-availability support.

Programmatic access to the API

Kubernetes officially supports Go and Python client libraries.

Go client

- To get the library, run the following command: `go get k8s.io/client-go/<version number>/kubernetes`. See https://github.com/kubernetes/client-go to see which versions are supported.
- Write an application atop of the client-go clients. Note that client-go defines its own API objects, so if needed, please import API definitions from client-go rather than from the main repository, e.g., `import "k8s.io/client-go/1.4/pkg/api/v1"` is correct.

The Go client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the apiserver. See this example.

If the application is deployed as a Pod in the cluster, please refer to the next section.

Python client

To use Python client, run the following command: `pip install kubernetes`. See Python Client Library page for more installation options.

The Python client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the apiserver. See this example.

Other languages

There are client libraries for accessing the API from other languages. See documentation for other libraries for how they authenticate.

Accessing the API from a Pod

When accessing the API from a pod, locating and authenticating to the apiserver are somewhat different.

The recommended way to locate the apiserver within the pod is with the `kubernetes` DNS name, which resolves to a Service IP which in turn will be routed to an apiserver.

The recommended way to authenticate to the apiserver is with a service account credential. By kube-system, a pod is associated with a service account, and a credential (token) for that service account is placed into the filesystem tree of each container in that pod, at `/var/run/secrets/kubernetes.io/serviceaccount/token` .

If available, a certificate bundle is placed into the filesystem tree of each container at `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` , and should be used to verify the serving certificate of the apiserver.

Finally, the default namespace to be used for namespaced API operations is placed in a file at `/var/run/secrets/kubernetes.io/serviceaccount/namespace` in each container.

From within a pod the recommended ways to connect to API are:

- run a kubectl proxy as one of the containers in the pod, or as a background process within a container. This proxies the Kubernetes API to the localhost interface of the pod, so that other processes in any container of the pod can access it. See this example of using kubectl proxy in a pod.
- use the Go client library, and create a client using the `rest.InClusterConfig()` and `kubernetes.NewForConfig()` functions. They handle locating and authenticating to the apiserver. example

In each case, the credentials of the pod are used to communicate securely with the apiserver.

Accessing services running on the cluster

The previous section was about connecting the Kubernetes API server. This section is about connecting to other services running on Kubernetes cluster. In Kubernetes, the nodes, pods and services all have their own IPs. In many cases, the node IPs, pod IPs, and some service IPs on a cluster will not be routable, so they will not be reachable from a machine outside the cluster, such as your desktop machine.

Ways to connect

You have several options for connecting to nodes, pods and services from outside the cluster:

- Access services through public IPs.
  - Use a service with type `NodePort` or `LoadBalancer` to make the service reachable outside the cluster. See the services and kubectl expose documentation.
  - Depending on your cluster environment, this may just expose the service to your corporate network, or it may expose it to the internet. Think about whether the service being exposed is secure. Does it do its own authentication?
  - Place pods behind services. To access one specific pod from a set of replicas, such as for debugging, place a unique label on the pod and create a new service which selects this label.
  - In most cases, it should not be necessary for application developer to directly access nodes via their nodeIPs.
- Access services, nodes, or pods using the Proxy Verb.
  - Does apiserver authentication and authorization prior to accessing the remote service. Use this if the services are not secure enough to expose to the internet, or to gain access to ports on the node IP, or for debugging.
  - Proxies may cause problems for some web applications.
  - Only works for HTTP/HTTPS.
  - Described here.
- Access from a node or pod in the cluster.

- Run a pod, and then connect to a shell in it using kubectl exec. Connect to other nodes, pods, and services from that shell.
- Some clusters may allow you to ssh to a node in the cluster. From there you may be able to access cluster services. This is a non-standard method, and will work on some clusters but not others. Browsers and other tools may or may not be installed. Cluster DNS may not work.

Discovering builtin services

Typically, there are several services which are started on a cluster by kube-system. Get a list of these with the `kubectl cluster-info` command:

```
$ kubectl cluster-info

  Kubernetes master is running at https://104.197.5.247
  elasticsearch-logging is running at https://104.197.5.247/api/v1/namespaces/kube
  kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system
  kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/servi
  grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/servic
  heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/servi
```

This shows the proxy-verb URL for accessing each service. For example, this cluster has cluster-level logging enabled (using Elasticsearch), which can be reached at `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` if suitable credentials are passed. Logging can also be reached through a kubectl proxy, for example at: `http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`. (See above for how to pass credentials or use kubectl proxy.)

Manually constructing apiserver proxy URLs

As mentioned above, you use the `kubectl cluster-info` command to retrieve the service's proxy URL. To create proxy URLs that include service endpoints, suffixes, and parameters, you simply append to the service's proxy URL: `http://` *`kubernetes_master_address`* `/api/v1/namespaces/` *`namespace_name`* `/services/` *`service_name[:port_name]`* `/proxy`

If you haven't specified a name for your port, you don't have to specify *port_name* in the URL.

By default, the API server proxies to your service using http. To use https, prefix the service name with `https:` : `http://` *`kubernetes_master_address`* `/api/v1/namespaces/` *`namespace_name`* `/services/` *`https:service_name:[port_name]`* `/proxy`

The supported formats for the name segment of the URL are:

- `<service_name>` - proxies to the default or unnamed port using http
- `<service_name>:<port_name>` - proxies to the specified port using http
- `https:<service_name>:` - proxies to the default or unnamed port using https (note the trailing colon)
- `https:<service_name>:<port_name>` - proxies to the specified port using https

Examples

- To access the Elasticsearch service endpoint `_search?q=user:kimchy`, you would use: `http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy`
- To access the Elasticsearch cluster health information `_cluster/health?pretty=true`, you would use: `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_cluster/health?pretty=true`

```
{
  "cluster_name" : "kubernetes_logging",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

Using web browsers to access services running on the cluster

You may be able to put an apiserver proxy url into the address bar of a browser. However:

- Web browsers cannot usually pass tokens, so you may need to use basic (password) auth. Apiserver can be configured to accept basic auth, but your cluster may not be configured to accept basic auth.
- Some web apps may not work, particularly those with client side javascript that construct urls in a way that is unaware of the proxy path prefix.

Requesting redirects

The redirect capabilities have been deprecated and removed. Please use a proxy (see below) instead.

So Many Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The kubectl proxy: - runs on a user's desktop or in a pod - proxies from a localhost address to the Kubernetes apiserver - client to proxy uses HTTP - proxy to apiserver uses HTTPS - locates apiserver - adds authentication headers
2. The apiserver proxy: - is a bastion built into the apiserver - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable - runs in the apiserver processes - client to

proxy uses HTTPS (or http if apiserver so configured) - proxy to target may use HTTP or HTTPS as chosen by proxy using available information - can be used to reach a Node, Pod, or Service - does load balancing when used to reach a Service

3. The kube proxy: - runs on each node - proxies UDP and TCP - does not understand HTTP - provides load balancing - is just used to reach services

4. A Proxy/Load-balancer in front of apiserver(s): - existence and implementation varies from cluster to cluster (e.g. nginx) - sits between all clients and one or more apiservers - acts as load balancer if there are several apiservers.

5. Cloud Load Balancers on external services: - are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer) - are created automatically when the Kubernetes service has type `LoadBalancer` - use UDP/TCP only - implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

Configure Access to Multiple Clusters

This page shows how to configure access to multiple clusters by using configuration files. After your clusters, users, and contexts are defined in one or more configuration files, you can quickly switch between clusters by using the `kubectl config use-context` command.

> **Note:** A file that is used to configure access to a cluster is sometimes called a *kubeconfig file*. This is a generic way of referring to configuration files. It does not mean that there is a file named `kubeconfig`.

Before you begin

You need to have the `kubectl` command-line tool installed.

Define clusters, users, and contexts

Suppose you have two clusters, one for development work and one for scratch work. In the `development` cluster, your frontend developers work in a namespace called `frontend`, and your storage developers work in a namespace called `storage`. In your `scratch` cluster, developers work in the default namespace, or they create auxiliary namespaces as they see fit. Access to the development cluster requires authentication by certificate. Access to the scratch cluster requires authentication by username and password.

Create a directory named `config-exercise`. In your `config-exercise` directory, create a file named `config-demo` with this content:

```
apiVersion: v1
kind: Config
preferences: {}

clusters:
- cluster:
  name: development
- cluster:
  name: scratch

users:
- name: developer
- name: experimenter

contexts:
- context:
  name: dev-frontend
- context:
  name: dev-storage
- context:
  name: exp-scratch
```

A configuration file describes clusters, users, and contexts. Your `config-demo` file has the framework to describe two clusters, two users, and three contexts.

Go to your `config-exercise` directory. Enter these commands to add cluster details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-cluster development --server=https://1
kubectl config --kubeconfig=config-demo set-cluster scratch --server=https://5.6.7
```

Add user details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-credentials developer --client-certifi
kubectl config --kubeconfig=config-demo set-credentials experimenter --username=ex
```

Add context details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-context dev-frontend --cluster=develop
kubectl config --kubeconfig=config-demo set-context dev-storage --cluster=developm
kubectl config --kubeconfig=config-demo set-context exp-scratch --cluster=scratch
```

Open your `config-demo` file to see the added details. As an alternative to opening the `config-demo` file, you can use the `config view` command.

```
kubectl config --kubeconfig=config-demo view
```

The output shows the two clusters, two users, and three contexts:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
  name: development
- cluster:
    insecure-skip-tls-verify: true
    server: https://5.6.7.8
  name: scratch
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
- context:
    cluster: development
    namespace: storage
    user: developer
  name: dev-storage
- context:
    cluster: scratch
    namespace: default
    user: experimenter
  name: exp-scratch
current-context: ""
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    password: some-password
    username: exp
```

Each context is a triple (cluster, user, namespace). For example, the `dev-frontend` context says, Use the credentials of the `developer` user to access the `frontend` namespace of the `development` cluster.

Set the current context:

```
kubectl config --kubeconfig=config-demo use-context dev-frontend
```

Now whenever you enter a `kubectl` command, the action will apply to the cluster, and namespace listed in the `dev-frontend` context. And the command will use the credentials of the user listed in the `dev-frontend` context.

To see only the configuration information associated with the current context, use the `--minify` flag.

```
kubectl config --kubeconfig=config-demo view --minify
```

The output shows configuration information associated with the `dev-frontend` context:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
  name: development
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```

Now suppose you want to work for a while in the scratch cluster.

Change the current context to `exp-scratch`:

```
kubectl config --kubeconfig=config-demo use-context exp-scratch
```

Now any `kubectl` command you give will apply to the default namespace of the `scratch` cluster. And the command will use the credentials of the user listed in the `exa-scratch` context.

View configuration associated with the new current context, `exp-scratch`.

```
kubectl config --kubeconfig=config-demo view --minify
```

Finally, suppose you want to work for a while in the `storage` namespace of the `development` cluster.

Change the current context to `dev-storage`:

```
kubectl config --kubeconfig=config-demo use-context dev-storage
```

View configuration associated with the new current context, `dev-storage.

```
kubectl config --kubeconfig=config-demo view --minify
```

Create a second configuration file

In your `config-exercise` directory, create a file named `config-demo-2` with this content:

```
apiVersion: v1
kind: Config
preferences: {}

contexts:
- context:
    cluster: development
    namespace: ramp
    user: developer
  name: dev-ramp-up
```

The preceding configuration file defines a new context named `dev-ramp-up`.

Set the KUBECONFIG environment variable

See whether you have an environment variable named `KUBECONFIG`. If so, save the current value of your `KUBECONFIG` environment variable, so you can restore it later. For example, on Linux:

```
export KUBECONFIG_SAVED=$KUBECONFIG
```

The `KUBECONFIG` environment variable is a list of paths to configuration files. The list is colon-delimited for Linux and Mac, and semicolon-delimited for Windows. If you have a `KUBECONFIG` environment variable, familiarize yourself with the configuration files in the list.

Temporarily append two paths to your `KUBECONFIG` environment variable. For example, on Linux:

```
export KUBECONFIG=$KUBECONFIG:config-demo:config-demo-2
```

In your `config-exercise` directory, enter this command:

```
kubectl config view
```

The output shows merged information from all the files listed in your `KUBECONFIG` environment variable. In particular, notice that the merged information has the `dev-ramp-up` context from the `config-demo-2` file and the three contexts from the `config-demo` file:

```
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
- context:
    cluster: development
    namespace: ramp
    user: developer
  name: dev-ramp-up
- context:
    cluster: development
    namespace: storage
    user: developer
  name: dev-storage
- context:
    cluster: scratch
    namespace: default
    user: experimenter
  name: exp-scratch
```

For more information about how kubeconfig files are merged, see Organizing Cluster Access Using kubeconfig Files

Explore the $HOME/.kube directory

If you already have a cluster, and you can use `kubectl` to interact with the cluster, then you probably have a file named `config` in the `$HOME/.kube` directory.

Go to `$HOME/.kube`, and see what files are there. Typically, there is a file named `config`. There might also be other configuration files in this directory. Briefly familiarize yourself with the contents of these files.

Append $HOME/.kube/config to your KUBECONFIG environment variable

If you have a `$HOME/.kube/config` file, and it's not already listed in your `KUBECONFIG` environment variable, append it to your `KUBECONFIG` environment variable now. For example, on Linux:

```
export KUBECONFIG=$KUBECONFIG:$HOME/.kube/config
```

View configuration information merged from all the files that are now listed in your `KUBECONFIG` environment variable. In your config-exercise directory, enter:

```
kubectl config view
```

Clean up

Return your `KUBECONFIG` environment variable to its original value. For example, on Linux:

```
export KUBECONFIG=$KUBECONFIG_SAVED
```

What's next

- Organizing Cluster Access Using kubeconfig Files
- kubectl config

Use Port Forwarding to Access Applications in a Cluster

This page shows how to use `kubectl port-forward` to connect to a Redis server running in a Kubernetes cluster. This type of connection can be useful for database debugging.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- Install redis-cli.

Creating a pod to run a Redis server

1. Create a pod:

```
kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/redis-ma
```

The output of a successful command verifies that the pod was created:

```
pod "redis-master" created
```

2. Check to see whether the pod is running and ready:

```
kubectl get pods
```

When the pod is ready, the output displays a STATUS of Running:

```
NAME           READY     STATUS    RESTARTS   AGE
redis-master   2/2       Running   0          41s
```

3. Verify that the Redis server is running in the pod and listening on port 6379:

```
kubectl get pods redis-master --template='{{(index (index .spec.containers 0).
```

The output displays the port:

```
6379
```

Forward a local port to a port on the pod

1. Forward port 6379 on the local workstation to port 6379 of redis-master pod:

```
kubectl port-forward redis-master 6379:6379
```

The output is similar to this:

```
I0710 14:43:38.274550    3655 portforward.go:225] Forwarding from 127.0.0.1:6
I0710 14:43:38.274797    3655 portforward.go:225] Forwarding from [::1]:6379
```

2. Start the Redis command line interface:

```
redis-cli
```

3. At the Redis command line prompt, enter the `ping` command:

```
127.0.0.1:6379>ping
```

A successful ping request returns PONG.

Discussion

Connections made to local port 6379 are forwarded to port 6379 of the pod that is running the Redis server. With this connection in place you can use your local workstation to debug the database that is running in the pod.

What's next

Learn more about kubectl port-forward.

Provide Load-Balanced Access to an Application in a Cluster

This page shows how to create a Kubernetes Service object that provides load-balanced access to an application running in a cluster.

Objectives

- Run two instances of a Hello World application
- Create a Service object
- Use the Service object to access the running application

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Creating a Service for an application running in two pods

1. Run a Hello World application in your cluster:

   ```
   kubectl run hello-world --replicas=2 --labels="run=load-balancer-example" --ima
   ```

2. List the pods that are running the Hello World application:

   ```
   kubectl get pods --selector="run=load-balancer-example"
   ```

   The output is similar to this:

   ```
   NAME                           READY     STATUS    RESTARTS   AGE
   hello-world-2189936611-8fyp0   1/1       Running   0          6m
   hello-world-2189936611-9isq8   1/1       Running   0          6m
   ```

3. List the replica set for the two Hello World pods:

   ```
   kubectl get replicasets --selector="run=load-balancer-example"
   ```

   The output is similar to this:

   ```
   NAME                     DESIRED   CURRENT   AGE
   hello-world-2189936611   2         2         12m
   ```

4. Create a Service object that exposes the replica set:

   ```
   kubectl expose rs <your-replica-set-name> --type="LoadBalancer" --name="exampl
   ```

   where `<your-replica-set-name>` is the name of your replica set.

5. Display the IP addresses for your service:

```
kubectl get services example-service
```

The output shows the internal IP address and the external IP address of your service. If the external IP address shows as `<pending>`, repeat the command.

Note: If you are using Minikube, you don't get an external IP address. The external IP address remains in the pending state.

```
NAME              CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
example-service   10.0.0.160   <pending>     8080/TCP   40s
```

6. Use your Service object to access the Hello World application:

```
curl <your-external-ip-address>:8080
```

where `<your-external-ip-address>` is the external IP address of your service.

The output is a hello message from the application:

```
Hello Kubernetes!
```

Note: If you are using Minikube, enter these commands:

```
kubectl cluster-info
kubectl describe services example-service
```

The output displays the IP address of your Minikube node and the NodePort value for your service. Then enter this command to access the Hello World application:

```
curl <minikube-node-ip-address>:<service-node-port>
```

where `<minikube-node-ip-address>` us the IP address of your Minikube node, and `<service-node-port>` is the NodePort value for your service.

Using a service configuration file

As an alternative to using `kubectl expose`, you can use a service configuration file to create a Service.

What's next

Learn more about connecting applications with services.

Connect a Front End to a Back End Using a Service

This task shows how to create a frontend and a backend microservice. The backend microservice is a hello greeter. The frontend and backend are connected using a Kubernetes Service object.

Objectives

- Create and run a microservice using a Deployment object.
- Route traffic to the backend using a frontend.
- Use a Service object to connect the frontend application to the backend application.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes

- This task uses Services with external load balancers, which require a supported environment. If your environment does not support this, you can use a Service of type NodePort instead.

Creating the backend using a Deployment

The backend is a simple hello greeter microservice. Here is the configuration file for the backend Deployment:

hello.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 7
  template:
    metadata:
      labels:
        app: hello
        tier: backend
        track: stable
    spec:
      containers:
        - name: hello
          image: "gcr.io/google-samples/hello-go-gke:1.0"
          ports:
            - name: http
              containerPort: 80
```

Create the backend Deployment:

```
kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/hello.yaml
```

View information about the backend Deployment:

```
kubectl describe deployment hello
```

The output is similar to this:

```
Name:                           hello
Namespace:                      default
CreationTimestamp:              Mon, 24 Oct 2016 14:21:02 -0700
Labels:                         app=hello
                                tier=backend
                                track=stable
Annotations:                    deployment.kubernetes.io/revision=1
Selector:                       app=hello,tier=backend,track=stable
Replicas:                       7 desired | 7 updated | 7 total | 7 available | 0
StrategyType:                   RollingUpdate
MinReadySeconds:                0
RollingUpdateStrategy:          1 max unavailable, 1 max surge
Pod Template:
  Labels:       app=hello
                tier=backend
                track=stable
  Containers:
   hello:
    Image:              "gcr.io/google-samples/hello-go-gke:1.0"
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:                 <none>
NewReplicaSet:                  hello-3621623197 (7/7 replicas created)
Events:
...
```

Creating the backend Service object

The key to connecting a frontend to a backend is the backend Service. A Service creates a persistent IP address and DNS name entry so that the backend microservice can always be reached. A Service uses selector labels to find the Pods that it routes traffic to.

First, explore the Service configuration file:

hello-service.yaml

hello-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: hello
spec:
  selector:
    app: hello
    tier: backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: http
```

In the configuration file, you can see that the Service routes traffic to Pods that have the labels `app: hello` and `tier: backend`.

Create the `hello` Service:

```
kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/hello-servi
```

At this point, you have a backend Deployment running, and you have a Service that can route traffic to it.

Creating the frontend

Now that you have your backend, you can create a frontend that connects to the backend. The frontend connects to the backend worker Pods by using the DNS name given to the backend Service. The DNS name is “hello”, which is the value of the `name` field in the preceding Service configuration file.

The Pods in the frontend Deployment run an nginx image that is configured to find the hello backend Service. Here is the nginx configuration file:

frontend/frontend.conf

frontend/frontend.conf

```
upstream hello {
    server hello;
}

server {
    listen 80;

    location / {
        proxy_pass http://hello;
    }
}
```

Similar to the backend, the frontend has a Deployment and a Service. The configuration for the Service has `type: LoadBalancer`, which means that the Service uses the default load balancer of your cloud provider.

frontend.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: frontend
spec:
  selector:
    app: hello
    tier: frontend
  ports:
  - protocol: "TCP"
    port: 80
    targetPort: 80
  type: LoadBalancer
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hello
        tier: frontend
        track: stable
    spec:
      containers:
      - name: nginx
        image: "gcr.io/google-samples/hello-frontend:1.0"
        lifecycle:
          preStop:
            exec:
              command: ["/usr/sbin/nginx","-s","quit"]
```

Create the frontend Deployment and Service:

```
kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/frontend.ya
```

The output verifies that both resources were created:

```
deployment "frontend" created
service "frontend" created
```

Note: The nginx configuration is baked into the container image. A better way to do this would be to use a ConfigMap, so that you can change the configuration more easily.

Interact with the frontend Service

Once you've created a Service of type LoadBalancer, you can use this command to find the external IP:

```
kubectl get service frontend
```

The external IP field may take some time to populate. If this is the case, the external IP is listed as `<pending>` .

```
NAME       CLUSTER-IP      EXTERNAL-IP   PORT(S)  AGE
frontend   10.51.252.116   <pending>     80/TCP   10s
```

Repeat the same command again until it shows an external IP address:

```
NAME       CLUSTER-IP      EXTERNAL-IP        PORT(S)  AGE
frontend   10.51.252.116   XXX.XXX.XXX.XXX    80/TCP   1m
```

Send traffic through the frontend

The frontend and backends are now connected. You can hit the endpoint by using the curl command on the external IP of your frontend Service.

```
curl http://<EXTERNAL-IP>
```

The output shows the message generated by the backend:

```
{"message":"Hello"}
```

What's next

- Learn more about Services
- Learn more about ConfigMaps

Create an External Load Balancer

This page shows how to create an External Load Balancer.

When creating a service, you have the option of automatically creating a cloud network load balancer. This provides an externally-accessible IP address that sends traffic to the correct port on your cluster nodes *provided your cluster runs in a supported environment and is configured with the correct cloud load balancer provider package*.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda

- Play with Kubernetes

Configuration file

To create an external load balancer, add the following line to your service configuration file:

```
    "type": "LoadBalancer"
```

Your configuration file might look like:

```
    {
      "kind": "Service",
      "apiVersion": "v1",
      "metadata": {
        "name": "example-service"
      },
      "spec": {
        "ports": [{
          "port": 8765,
          "targetPort": 9376
        }],
        "selector": {
          "app": "example"
        },
        "type": "LoadBalancer"
      }
    }
```

Using kubectl

You can alternatively create the service with the `kubectl expose` command and its `--type=LoadBalancer` flag:

```
kubectl expose rc example --port=8765 --target-port=9376 \
        --name=example-service --type=LoadBalancer
```

This command creates a new service using the same selectors as the referenced resource (in the case of the example above, a replication controller named `example`).

For more information, including optional flags, refer to the `kubectl expose` reference.

Finding your IP address

You can find the IP address created for your service by getting the service information through `kubectl`:

```
kubectl describe services example-service
```

which should produce output like this:

```
    Name:                   example-service
    Namespace:              default
    Labels:                 <none>
    Annotations:            <none>
    Selector:               app=example
    Type:                   LoadBalancer
    IP:                     10.67.252.103
    LoadBalancer Ingress:   123.45.678.9
    Port:                   <unnamed> 80/TCP
    NodePort:               <unnamed> 32445/TCP
    Endpoints:              10.64.0.4:80,10.64.1.5:80,10.64.2.4:80
    Session Affinity:       None
    Events:                 <none>
```

The IP address is listed next to `LoadBalancer Ingress`.

Preserving the client source IP

Due to the implementation of this feature, the source IP seen in the target container will *not be the original source IP* of the client. To enable preservation of the client IP, the following fields can be configured in the service spec (supported in GCE/GKE environments):

- `service.spec.externalTrafficPolicy` - denotes if this Service desires to route external traffic to node-local or cluster-wide endpoints. There are two available options: “Cluster” (default) and “Local”. “Cluster” obscures the client source IP and may cause a second hop to another node, but should have good overall load-spreading. “Local” preserves the client source IP and avoids a second hop for LoadBalancer and NodePort type services, but risks potentially imbalanced traffic spreading.
- `service.spec.healthCheckNodePort` - specifies the healthcheck nodePort (numeric port number) for the service. If not specified, healthCheckNodePort is created by the service API backend with the allocated nodePort. It will use the user-specified nodePort value if specified by

the client. It only has an effect when type is set to "LoadBalancer" and externalTrafficPolicy is set to "Local".

This feature can be activated by setting `externalTrafficPolicy` to "Local" in the Service Configuration file.

```
    {
      "kind": "Service",
      "apiVersion": "v1",
      "metadata": {
        "name": "example-service"
      },
      "spec": {
        "ports": [{
          "port": 8765,
          "targetPort": 9376
        }],
        "selector": {
          "app": "example"
        },
        "type": "LoadBalancer",
        "externalTrafficPolicy": "Local"
      }
    }
```

Feature availability

k8s version	Feature support
1.7+	Supports the full API fields
1.5 - 1.6	Supports Beta Annotations
<1.5	Unsupported

Below you could find the deprecated Beta annotations used to enable this feature prior to its stable version. Newer Kubernetes versions may stop supporting these after v1.7. Please update existing applications to use the fields directly.

- `service.beta.kubernetes.io/external-traffic` annotation <-> `service.spec.externalTrafficPolicy` field

- `service.beta.kubernetes.io/healthcheck-nodeport` annotation <-> `service.spec.healthCheckNodePort` field

`service.beta.kubernetes.io/external-traffic` annotation has a different set of values compared to the `service.spec.externalTrafficPolicy` field. The values match as follows:

- "OnlyLocal" for annotation <-> "Local" for field
- "Global" for annotation <-> "Cluster" for field

Note that this feature is not currently implemented for all cloudproviders/environments.

Known issues:

- AWS: kubernetes/kubernetes#35758
- Weave-Net: weaveworks/weave/#2924

External Load Balancer Providers

It is important to note that the datapath for this functionality is provided by a load balancer external to the Kubernetes cluster.

When the service type is set to `LoadBalancer`, Kubernetes provides functionality equivalent to `type=<ClusterIP>` to pods within the cluster and extends it by programming the (external to Kubernetes) load balancer with entries for the Kubernetes VMs. The Kubernetes service controller automates the creation of the external load balancer, health checks (if needed), firewall rules (if needed) and retrieves the external IP allocated by the cloud provider and populates it in the service object.

Caveats and Limitations when preserving source IPs

GCE/AWS load balancers do not provide weights for their target pools. This was not an issue with the old LB kube-proxy rules which would correctly balance across all endpoints.

With the new functionality, the external traffic will not be equally load balanced across pods, but rather equally balanced at the node level (because GCE/AWS and other external LB implementations do not have the ability for specifying the weight per node, they balance equally across all target nodes, disregarding the number of pods on each node).

We can, however, state that for NumServicePods « NumNodes or NumServicePods » NumNodes, a fairly close-to-equal distribution will be seen, even without weights.

Once the external load balancers provide weights, this functionality can be added to the LB programming path. *Future Work: No support for weights is provided for the 1.4 release, but may be added at a future date*

Internal pod to pod traffic should behave similar to ClusterIP services, with equal probability across all pods.

Configure Your Cloud Provider's Firewalls

Many cloud providers (e.g. Google Compute Engine) define firewalls that help prevent inadvertent exposure to the internet. When exposing a service to the external world, you may need to open up one or more ports in these firewalls to serve traffic. This document describes this process, as well as any provider specific details that may be necessary.

Restrict Access For LoadBalancer Service

When using a Service with `spec.type: LoadBalancer`, you can specify the IP ranges that are allowed to access the load balancer by using `spec.loadBalancerSourceRanges`. This field takes a list of IP CIDR ranges, which Kubernetes will use to configure firewall exceptions. This feature is currently supported on Google Compute Engine, Google Container Engine and AWS. This field will be ignored if the cloud provider does not support the feature.

Assuming 10.0.0.0/8 is the internal subnet. In the following example, a load balancer will be created that is only accessible to cluster internal IPs. This will not allow clients from outside of your Kubernetes cluster to access the load balancer.

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  ports:
  - port: 8765
    targetPort: 9376
  selector:
    app: example
  type: LoadBalancer
  loadBalancerSourceRanges:
  - 10.0.0.0/8
```

In the following example, a load balancer will be created that is only accessible to clients with IP addresses from 130.211.204.1 and 130.211.204.2.

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  ports:
  - port: 8765
    targetPort: 9376
  selector:
    app: example
  type: LoadBalancer
  loadBalancerSourceRanges:
  - 130.211.204.1/32
  - 130.211.204.2/32
```

Google Compute Engine

When using a Service with `spec.type: LoadBalancer`, the firewall will be opened automatically. When using `spec.type: NodePort`, however, the firewall is *not* opened by default.

Google Compute Engine firewalls are documented elsewhere.

You can add a firewall with the `gcloud` command line tool:

```
$ gcloud compute firewall-rules create my-rule --allow=tcp:<port>
```

Note There is one important security note when using firewalls on Google Compute Engine:

as of Kubernetes v1.0.0, GCE firewalls are defined per-vm, rather than per-ip address. This means that when you open a firewall for a service's ports, anything that serves on that port on that VM's host IP address may potentially serve traffic. Note that this is not a problem for other Kubernetes services, as they listen on IP addresses that are different than the host node's external IP address.

Consider:

- You create a Service with an external load balancer (IP Address 1.2.3.4) and port 80
- You open the firewall for port 80 for all nodes in your cluster, so that the external Service actually can deliver packets to your Service
- You start an nginx server, running on port 80 on the host virtual machine (IP Address 2.3.4.5). This nginx is also exposed to the internet on the VM's external IP address.

Consequently, please be careful when opening firewalls in Google Compute Engine or Google Container Engine. You may accidentally be exposing other services to the wilds of the internet.

This will be fixed in an upcoming release of Kubernetes.

Other cloud providers

Coming soon.

List All Container Images Running in a Cluster

This page shows how to use kubectl to list all of the Container images for Pods running in a cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

In this exercise you will use kubectl to fetch all of the Pods running in a cluster, and format the output to pull out the list of Containers for each.

List all Containers in all namespaces

- Fetch all Pods in all namespaces using `kubectl get pods --all-namespaces`
- Format the output to include only the list of Container image names using `-o jsonpath={..image}`. This will recursively parse out the `image` field from the returned

json.

  - See the jsonpath reference for further information on how to use jsonpath.

- Format the output using standard tools: `tr`, `sort`, `uniq`
  - Use `tr` to replace spaces with newlines
  - Use `sort` to sort the results
  - Use `uniq` to aggregate image counts

```
kubectl get pods --all-namespaces -o jsonpath="{..image}" |\
tr -s '[[:space:]]' '\n' |\
sort |\
uniq -c
```

The above command will recursively return all fields named `image` for all items returned.

As an alternative, it is possible to use the absolute path to the image field within the Pod. This ensures the correct field is retrieved even when the field name is repeated, e.g. many fields are called `name` within a given item:

```
kubectl get pods --all-namespaces -o jsonpath="{.items[*].spec.containers[*].image
```

The jsonpath is interpreted as follows:

- `.items[*]`: for each returned value
- `.spec`: get the spec
- `.containers[*]`: for each container
- `.image`: get the image

Note: When fetching a single Pod by name, e.g. `kubectl get pod nginx`, the `.items[*]` portion of the path should be omitted because a single Pod is returned instead of a list of items.

List Containers by Pod

The formatting can be controlled further by using the `range` operation to iterate over elements individually.

```
kubectl get pods --all-namespaces -o=jsonpath='{range .items[*]}{"\n"}{.metadata.n
sort
```

List Containers filtering by Pod label

To target only Pods matching a specific label, use the -l flag. The following matches only Pods with labels matching `app=nginx`.

```
kubectl get pods --all-namespaces -o=jsonpath="{..image}" -l app=nginx
```

List Containers filtering by Pod namespace

To target only pods in a specific namespace, use the namespace flag. The following matches only Pods in the `kube-system` namespace.

```
kubectl get pods --namespace kube-system -o jsonpath="{..image}"
```

List Containers using a go-template instead of jsonpath

As an alternative to jsonpath, Kubectl supports using go-templates for formatting the output:

```
kubectl get pods --all-namespaces -o go-template --template="{{range .items}}{{ran
```

What's next

Reference

- Jsonpath reference guide
- Go template reference guide

Communicate Between Containers in the Same Pod Using a Shared Volume

This page shows how to use a Volume to communicate between two Containers running in the same Pod.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Creating a Pod that runs two Containers

In this exercise, you create a Pod that runs two Containers. The two containers share a Volume that they can use to communicate. Here is the configuration file for the Pod:

two-container-pod.yaml

two-container-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:

  restartPolicy: Never

  volumes:
  - name: shared-data
    emptyDir: {}

  containers:

  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html

  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the debian container > /pod-data/index.html"]
```

In the configuration file, you can see that the Pod has a Volume named `shared-data`.

The first container listed in the configuration file runs an nginx server. The mount path for the shared Volume is `/usr/share/nginx/html`. The second container is based on the debian image, and has a mount path of `/pod-data`. The second container runs the following command and then terminates.

```
echo Hello from the debian container > /pod-data/index.html
```

Notice that the second container writes the `index.html` file in the root directory of the nginx server.

Create the Pod and the two Containers:

```
kubectl create -f https://k8s.io/docs/tasks/access-application-cluster/two-contain
```

View information about the Pod and the Containers:

```
kubectl get pod two-containers --output=yaml
```

Here is a portion of the output:

```
apiVersion: v1
kind: Pod
metadata:
  ...
  name: two-containers
  namespace: default
  ...
spec:
  ...
  containerStatuses:

  - containerID: docker://c1d8abd1 ...
    image: debian
    ...
    lastState:
      terminated:
        ...
    name: debian-container
    ...

  - containerID: docker://96c1ff2c5bb ...
    image: nginx
    ...
    name: nginx-container
    ...
    state:
      running:
    ...
```

You can see that the debian Container has terminated, and the nginx Container is still running.

Get a shell to nginx Container:

```
kubectl exec -it two-containers -c nginx-container -- /bin/bash
```

In your shell, verify that nginx is running:

```
root@two-containers:/# ps aux
```

The output is similar to this:

```
USER       PID  ...  STAT START   TIME COMMAND
root         1  ...  Ss   21:12   0:00 nginx: master process nginx -g daemon off;
```

Recall that the debian Container created the `index.html` file in the nginx root directory. Use `curl` to send a GET request to the nginx server:

```
root@two-containers:/# apt-get update
root@two-containers:/# apt-get install curl
root@two-containers:/# curl localhost
```

The output shows that nginx serves a web page written by the debian container:

```
Hello from the debian container
```

Discussion

The primary reason that Pods can have multiple containers is to support helper applications that assist a primary application. Typical examples of helper applications are data pullers, data pushers, and proxies. Helper and primary applications often need to communicate with each other. Typically this is done through a shared filesystem, as shown in this exercise, or through the loopback network interface, localhost. An example of this pattern is a web server along with a helper program that polls a Git repository for new updates.

The Volume in this exercise provides a way for Containers to communicate during the life of the Pod. If the Pod is deleted and recreated, any data stored in the shared Volume is lost.

What's next

- Learn more about patterns for composite containers.

- Learn about composite containers for modular architecture.
- See Configuring a Pod to Use a Volume for Storage.
- See Volume.
- See Pod.

Core metrics pipeline

Starting from Kubernetes 1.8, resource usage metrics, such as container CPU and memory usage, are available in Kubernetes through the Metrics API. These metrics can be either accessed directly by user, for example by using `kubectl top` command, or used by a controller in the cluster, e.g. Horizontal Pod Autoscaler, to make decisions.

The Metrics API

Through the Metrics API you can get the amount of resource currently used by a given node or a given pod. This API doesn't store the metric values, so it's not possible for example to get the amount of resources used by a given node 10 minutes ago.

The API no different from any other API:

- it is discoverable through the same endpoint as the other Kubernetes APIs under `/apis/metrics.k8s.io/` path
- it offers the same security, scalability and reliability guarantees

The API is defined in k8s.io/metrics repository. You can find more information about the API there.

Note: The API requires metrics server to be deployed in the cluster. Otherwise it will be not available.

Metrics Server

Metrics Server is a cluster-wide aggregator of resource usage data. Starting from Kubernetes 1.8 it's deployed by default in clusters created by `kube-up.sh` script as a Deployment object. If you use a different Kubernetes setup mechanism you can deploy it using the provided deployment yamls. It's supported in Kubernetes 1.7+ (see details below).

Metric server collects metrics from the Summary API, exposed by Kubelet on each node.

Metrics Server registered in the main API server through Kubernetes aggregator, which was introduced in Kubernetes 1.7.

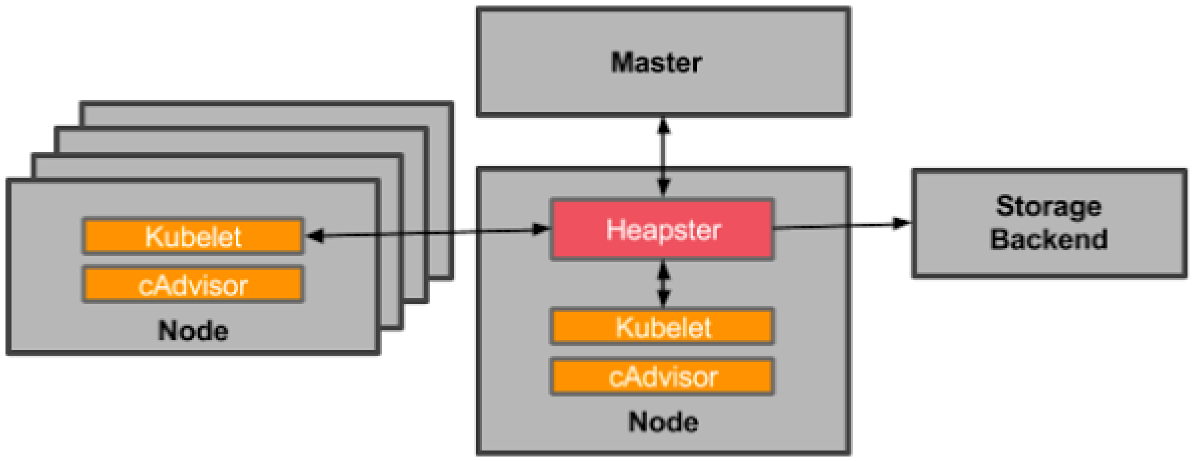
Learn more about the metrics server in the design doc.

Tools for Monitoring Compute, Storage, and Network Resources

Understanding how an application behaves when deployed is crucial to scaling the application and providing a reliable service. In a Kubernetes cluster, application performance can be examined at many different levels: containers, pods, services, and whole clusters. As part of Kubernetes we want to provide users with detailed resource usage information about their running applications at all these levels. This will give users deep insights into how their applications are performing and where possible application bottlenecks may be found. In comes Heapster, a project meant to provide a base monitoring platform on Kubernetes.

Overview

Heapster is a cluster-wide aggregator of monitoring and event data. It currently supports Kubernetes natively and works on all Kubernetes setups. Heapster runs as a pod in the cluster, similar to how any Kubernetes application would run. The Heapster pod discovers all nodes in the cluster and queries usage information from the nodes' Kubelets, the on-machine Kubernetes agent. The Kubelet itself fetches the data from cAdvisor. Heapster groups the information by pod along with the relevant labels. This data is then pushed to a configurable backend for storage and visualization. Currently supported backends include InfluxDB (with Grafana for visualization), Google Cloud Monitoring and many others described in more details here. The overall architecture of the service can be seen below:

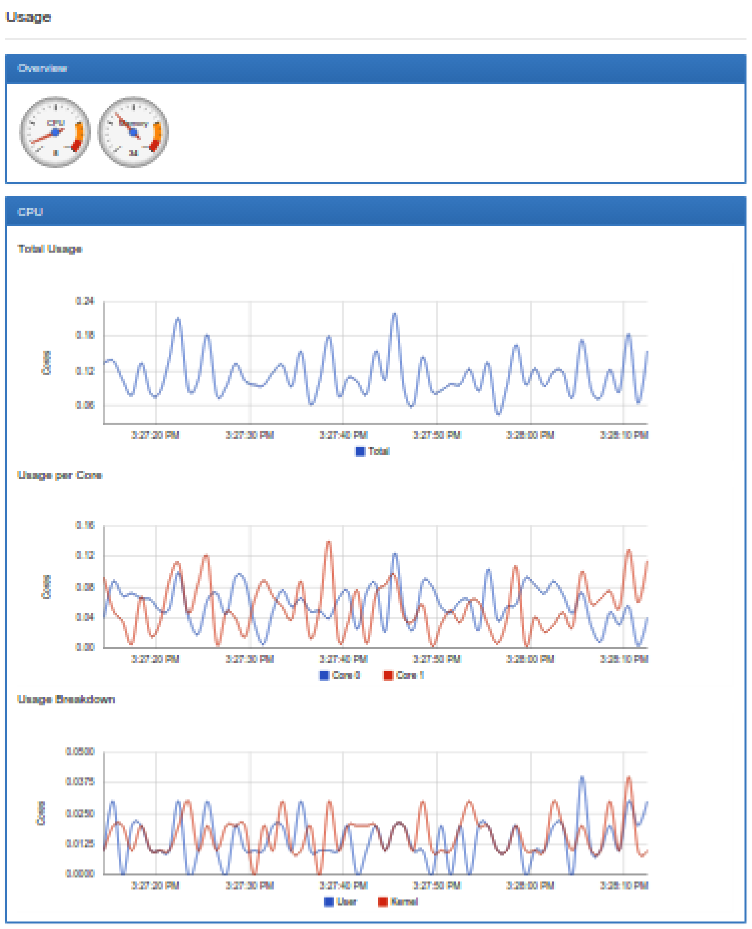


Let’s look at some of the other components in more detail.

cAdvisor

cAdvisor is an open source container resource usage and performance analysis agent. It is purpose-built for containers and supports Docker containers natively. In Kubernetes, cAdvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics. cAdvisor also provides the overall machine usage by analyzing the ‘root’ container on the machine.

On most Kubernetes clusters, cAdvisor exposes a simple UI for on-machine containers on port 4194. Here is a snapshot of part of cAdvisor’s UI that shows the overall machine usage:



Kubelet

The Kubelet acts as a bridge between the Kubernetes master and the nodes. It manages the pods and containers running on a machine. Kubelet translates each pod into its constituent containers and fetches individual container usage statistics from cAdvisor. It then exposes the aggregated pod resource usage statistics via a REST API.

Storage Backends

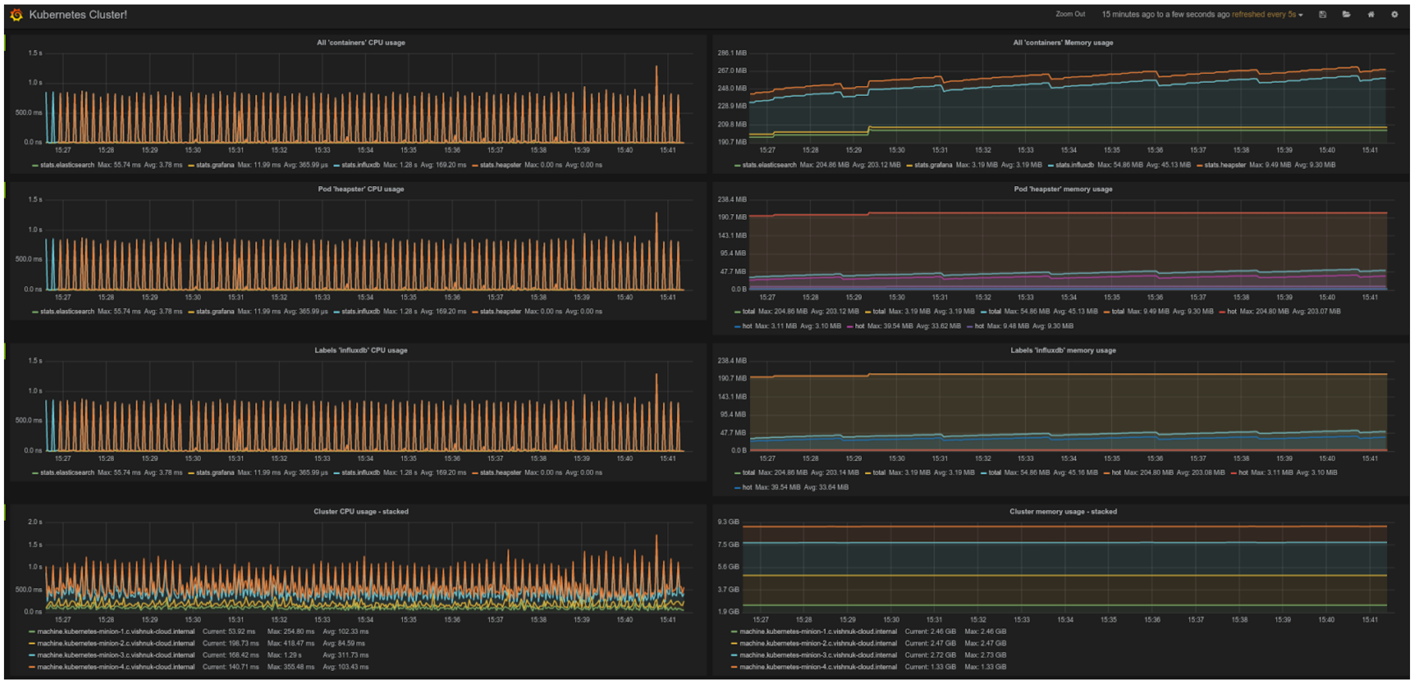
InfluxDB and Grafana

A Grafana setup with InfluxDB is a very popular combination for monitoring in the open source world. InfluxDB exposes an easy to use API to write and fetch time series data. Heapster is setup to use this storage backend by default on most Kubernetes clusters. A detailed setup guide can be found here. InfluxDB and Grafana run in Pods. The pod exposes itself as a Kubernetes service which is how Heapster discovers it.

The Grafana container serves Grafana's UI which provides an easy to configure dashboard interface. The default dashboard for Kubernetes contains an example dashboard that monitors resource usage of the cluster and the pods inside of it. This dashboard can easily be customized and expanded. Take a look at the storage schema for InfluxDB here.

Here is a video showing how to monitor a Kubernetes cluster using heapster, InfluxDB and Grafana:

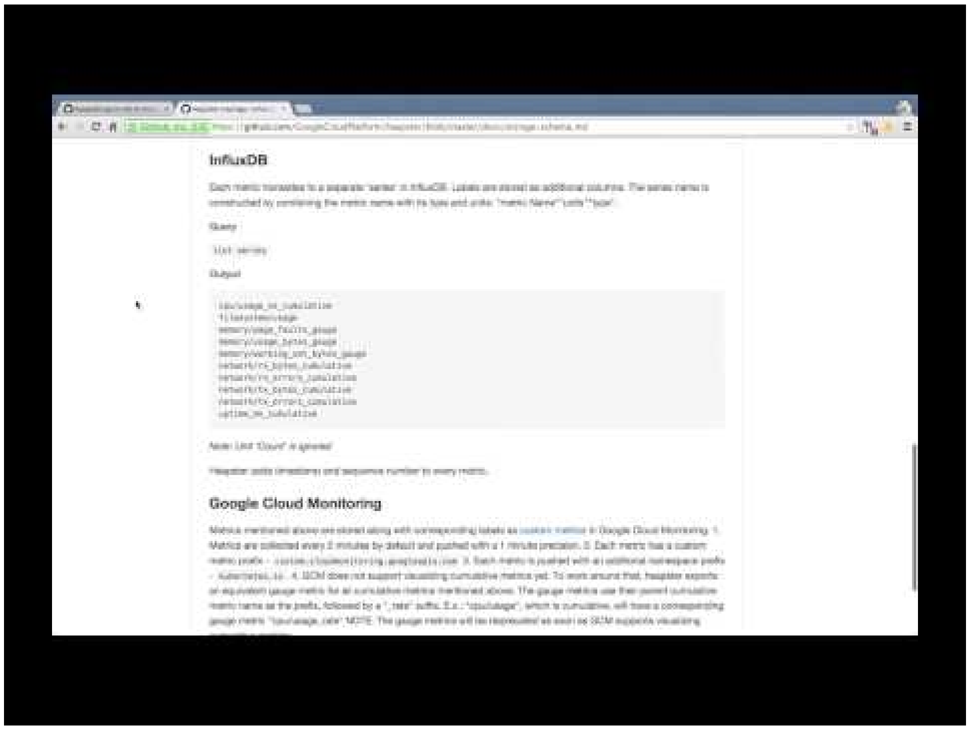
Here is a snapshot of the default Kubernetes Grafana dashboard that shows the CPU and Memory usage of the entire cluster, individual pods and containers:


Kubernetes Cluster!
All 'containers' CPU usage
All 'containers' Memory usage
Pod 'heapster' CPU usage
Pod 'heapster' memory usage
Labels 'influxdb' CPU usage
Labels 'influxdb' memory usage
Cluster CPU usage - stacked
Cluster memory usage - stacked

Google Cloud Monitoring

Google Cloud Monitoring is a hosted monitoring service that allows you to visualize and alert on important metrics in your application. Heapster can be setup to automatically push all collected metrics to Google Cloud Monitoring. These metrics are then available in the [Cloud Monitoring Console](). This storage backend is the easiest to setup and maintain. The monitoring console allows you to easily create and customize dashboards using the exported data.

Here is a video showing how to setup and run a Google Cloud Monitoring backed Heapster:



https://github.com/GoogleCloudPlatform/heapster/blob/master/docs/storage-schema.md

InfluxDB

Each metric translates to a separate "series" in InfluxDB. Labels are stored as additional columns. The series name is constructed by combining the metric name with its type and units: "metric Name"_"units"_"type".

Query

```
list series
```

Output

```
cpu/usage_ns_cumulative
filesystem/usage
memory/page_faults_gauge
memory/usage_bytes_gauge
memory/working_set_bytes_gauge
network/rx_bytes_cumulative
network/rx_errors_cumulative
network/tx_bytes_cumulative
network/tx_errors_cumulative
uptime_ms_cumulative
```

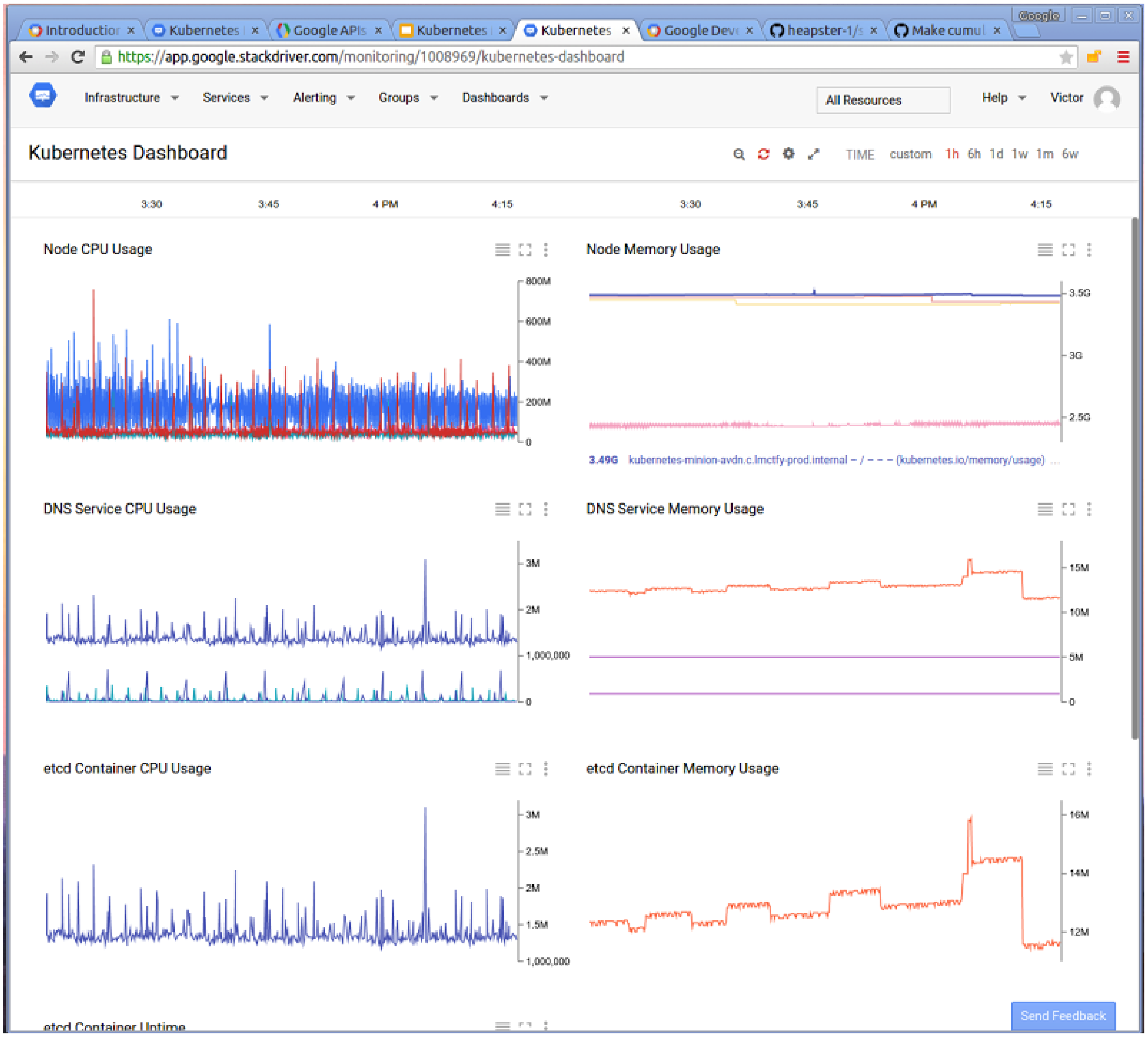
Note: Unit 'Count' is ignored.

Heapster adds timestamp and sequence number to every metric.

Google Cloud Monitoring

Metrics mentioned above are stored along with corresponding labels as custom metrics in Google Cloud Monitoring. 1. Metrics are collected every 2 minutes by default and pushed with a 1 minute precision. 2. Each metric has a custom metric prefix - `custom.cloudmonitoring.googleapis.com` 3. Each metric is pushed with an additional namespace prefix - `kubernetes.io`. 4. GCM does not support visualizing cumulative metrics yet. To work around that, heapster exports an equivalent gauge metric for all cumulative metrics mentioned above. The gauge metrics use their parent cumulative metric name as the prefix, followed by a "_rate" suffix. E.x.: "cpu/usage", which is cumulative, will have a corresponding gauge metric "cpu/usage_rate". NOTE: The gauge metrics will be deprecated as soon as GCM supports visualizing

Here is a snapshot of the Google Cloud Monitoring dashboard showing cluster-wide resource usage.



Try it out!

Now that you've learned a bit about Heapster, feel free to try it out on your own clusters! The Heapster repository is available on GitHub. It contains detailed instructions to setup Heapster and its storage backends. Heapster runs by default on most Kubernetes clusters, so you may already have it! Feedback is always welcome. Please let us know if you run into any issues via the troubleshooting channels.

Authors: Vishnu Kannan and Victor Marmol, Google Software Engineers. This article was originally posted in Kubernetes blog.

Get a Shell to a Running Container

This page shows how to use `kubectl exec` to get a shell to a running Container.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Getting a shell to a Container

In this exercise, you create a Pod that has one Container. The Container runs the nginx image. Here is the configuration file for the Pod:

shell-demo.yaml

shell-demo.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/shell-demo.y
```

Verify that the Container is running:

```
kubectl get pod shell-demo
```

Get a shell to the running Container:

```
kubectl exec -it shell-demo -- /bin/bash
```

In your shell, list the running processes:

```
root@shell-demo:/# ps aux
```

In your shell, list the nginx processes:

```
root@shell-demo:/# ps aux | grep nginx
```

In your shell, experiment with other commands. Here are some examples:

```
root@shell-demo:/# ls /
root@shell-demo:/# cat /proc/mounts
root@shell-demo:/# cat /proc/1/maps
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install tcpdump
root@shell-demo:/# tcpdump
root@shell-demo:/# apt-get install lsof
root@shell-demo:/# lsof
```

Writing the root page for nginx

Look again at the configuration file for your Pod. The Pod has an `emptyDir` volume, and the Container mounts the volume at `/usr/share/nginx/html`.

In your shell, create an `index.html` file in the `/usr/share/nginx/html` directory:

```
root@shell-demo:/# echo Hello shell demo > /usr/share/nginx/html/index.html
```

In your shell, send a GET request to the nginx server:

```
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install curl
root@shell-demo:/# curl localhost
```

The output shows the text that you wrote to the `index.html` file:

```
Hello shell demo
```

When you are finished with your shell, enter `exit`.

Running individual commands in a Container

In an ordinary command window, not your shell, list the environment variables in the running Container:

```
kubectl exec shell-demo env
```

Experiment running other commands. Here are some examples:

```
kubectl exec shell-demo ps aux
kubectl exec shell-demo ls /
kubectl exec shell-demo cat /proc/1/mounts
```

Opening a shell when a Pod has more than one Container

If a Pod has more than one Container, use `--container` or `-c` to specify a Container in the `kubectl exec` command. For example, suppose you have a Pod named my-pod, and the Pod has two containers named main-app and helper-app. The following command would open a shell to the main-app Container.

```
kubectl exec -it my-pod --container main-app -- /bin/bash
```

What's next

- kubectl exec

Monitor Node Health

Node Problem Detector

Node problem detector is a DaemonSet monitoring the node health. It collects node problems from various daemons and reports them to the apiserver as NodeCondition and Event.

It supports some known kernel issue detection now, and will detect more and more node problems over time.

Currently Kubernetes won't take any action on the node conditions and events generated by node problem detector. In the future, a remedy system could be introduced to deal with node problems.

See more information here.

Limitations

- The kernel issue detection of node problem detector only supports file based kernel log now. It doesn't support log tools like journald.

- The kernel issue detection of node problem detector has assumption on kernel log format, and now it only works on Ubuntu and Debian. However, it is easy to extend it to support other log format.

Enable/Disable in GCE cluster

Node problem detector is running as a cluster addon enabled by default in the gce cluster.

You can enable/disable it by setting the environment variable `KUBE_ENABLE_NODE_PROBLEM_DETECTOR` before `kube-up.sh`.

Use in Other Environment

To enable node problem detector in other environment outside of GCE, you can use either `kubectl` or addon pod.

Kubectl

This is the recommended way to start node problem detector outside of GCE. It provides more flexible management, such as overwriting the default configuration to fit it into your environment or detect customized node problems.

- **Step 1:** Create `node-problem-detector.yaml`:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: gcr.io/google_containers/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
```

Notice that you should make sure the system log directory is right for your OS distro.

- **Step 2:** Start node problem detector with `kubectl`:

```
kubectl create -f node-problem-detector.yaml
```

Addon Pod

This is for those who have their own cluster bootstrap solution, and don't need to overwrite the default configuration. They could leverage the addon pod to further automate the deployment.

Just create `node-problem-detector.yaml`, and put it under the addon pods directory `/etc/kubernetes/addons/node-problem-detector` on master node.

Overwrite the Configuration

The default configuration is embedded when building the docker image of node problem detector.

However, you can use ConfigMap to overwrite it following the steps:

- **Step 1:** Change the config files in `config/`.
- **Step 2:** Create the ConfigMap `node-problem-detector-config` with `kubectl create configmap node-problem-detector-config --from-file=config/`.
- **Step 3:** Change the `node-problem-detector.yaml` to use the ConfigMap:

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: gcr.io/google_containers/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
        - name: config # Overwrite the config/ directory with ConfigMap volume
          mountPath: /config
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
      - name: config # Define ConfigMap volume
        configMap:
          name: node-problem-detector-config
```

- **Step 4:** Re-create the node problem detector with the new yaml file:

```
kubectl delete -f node-problem-detector.yaml # If you have a node-problem-detector
kubectl create -f node-problem-detector.yaml
```

Notice that this approach only applies to node problem detector started with `kubectl`.

For node problem detector running as cluster addon, because addon manager doesn't support ConfigMap, configuration overwriting is not supported now.

Kernel Monitor

Kernel Monitor is a problem daemon in node problem detector. It monitors kernel log and detects known kernel issues following predefined rules.

The Kernel Monitor matches kernel issues according to a set of predefined rule list in `config/kernel-monitor.json`. The rule list is extensible, and you can always extend it by overwriting the configuration.

Add New NodeConditions

To support new node conditions, you can extend the `conditions` field in `config/kernel-monitor.json` with new condition definition:

```
{
  "type": "NodeConditionType",
  "reason": "CamelCaseDefaultNodeConditionReason",
  "message": "arbitrary default node condition message"
}
```

Detect New Problems

To detect new problems, you can extend the `rules` field in `config/kernel-monitor.json` with new rule definition:

```
{
  "type": "temporary/permanent",
  "condition": "NodeConditionOfPermanentIssue",
  "reason": "CamelCaseShortReason",
  "message": "regexp matching the issue in the kernel log"
}
```

Change Log Path

Kernel log in different OS distros may locate in different path. The `log` field in `config/kernel-monitor.json` is the log path inside the container. You can always configure it to match your OS distro.

Support Other Log Format

Kernel monitor uses `Translator` plugin to translate kernel log the internal data structure. It is easy to implement a new translator for a new log format.

Caveats

It is recommended to run the node problem detector in your cluster to monitor the node health. However, you should be aware that this will introduce extra resource overhead on each node. Usually this is fine, because:

- The kernel log is generated relatively slowly.
- Resource limit is set for node problem detector.
- Even under high load, the resource usage is acceptable. (see benchmark result)

Logging Using Stackdriver

Before reading this page, it's highly recommended to familiarize yourself with the overview of logging in Kubernetes.

Note: By default, Stackdriver logging collects only your container's standard output and standard error streams. To collect any logs your application writes to a file (for example), see the sidecar approach in the Kubernetes logging overview.

Deploying

To ingest logs, you must deploy the Stackdriver Logging agent to each node in your cluster. The agent is a configured `fluentd` instance, where the configuration is stored in a `ConfigMap` and the instances are managed using a Kubernetes `DaemonSet`. The actual deployment of the `ConfigMap` and `DaemonSet` for your cluster depends on your individual cluster setup.

Deploying to a new cluster

Google Container Engine

Stackdriver is the default logging solution for clusters deployed on Google Container Engine. Stackdriver Logging is deployed to a new cluster by default unless you explicitly opt-out.

Other platforms

To deploy Stackdriver Logging on a *new* cluster that you're creating using `kube-up.sh`, do the following:

1. Set the `KUBE_LOGGING_DESTINATION` environment variable to `gcp`.
2. **If not running on GCE**, include the `beta.kubernetes.io/fluentd-ds-ready=true` in the `KUBE_NODE_LABELS` variable.

Once your cluster has started, each node should be running the Stackdriver Logging agent. The `DaemonSet` and `ConfigMap` are configured as addons. If you're not using `kube-up.sh`, consider starting a cluster without a pre-configured logging solution and then deploying Stackdriver Logging agents to the running cluster.

Deploying to an existing cluster

1. Apply a label on each node, if not already present.

   The Stackdriver Logging agent deployment uses node labels to determine to which nodes it should be allocated. These labels were introduced to distinguish nodes with the Kubernetes version 1.6 or higher. If the cluster was created with Stackdriver Logging configured and node has version 1.5.X or lower, it will have fluentd as static pod. Node cannot have more than one instance of fluentd, therefore only apply labels to the nodes that don't have fluentd pod allocated already. You can ensure that your node is labelled properly by running `kubectl describe` as follows:

   ```
   kubectl describe node $NODE_NAME
   ```

   The output should be similar to this:

   ```
   Name: NODE_NAME Role: Labels: beta.kubernetes.io/fluentd-ds-ready=true ...
   ```

   Ensure that the output contains the label `beta.kubernetes.io/fluentd-ds-ready=true`. If it is not present, you can add it using the `kubectl label` command as follows:

   ```
   kubectl label node $NODE_NAME beta.kubernetes.io/fluentd-ds-ready=true
   ```

   Note: If a node fails and has to be recreated, you must re-apply the label to the recreated node. To make this easier, you can use Kubelet's command-line parameter for applying node labels in your node startup script.

2. Deploy a `ConfigMap` with the logging agent configuration by running the following command:

   ```
   kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/fluentd-
   gcp-configmap.yaml
   ```

   The command creates the `ConfigMap` in the `default` namespace. You can download the file manually and change it before creating the `ConfigMap` object.

3. Deploy the logging agent `DaemonSet` by running the following command:

   ```
   kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/fluentd-gcp-ds.yaml
   ```

   You can download and edit this file before using it as well.

Verifying your Logging Agent Deployment

After Stackdriver `DaemonSet` is deployed, you can discover logging agent deployment status by running the following command:

```
kubectl get ds --all-namespaces
```

If you have 3 nodes in the cluster, the output should looks similar to this:

```
NAMESPACE     NAME               DESIRED   CURRENT   READY     NODE-SELECTOR
...
kube-system   fluentd-gcp-v2.0   3         3         3         beta.kubernetes.io/
...
```

To understand how logging with Stackdriver works, consider the following synthetic log generator pod specification counter-pod.yaml:

counter-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
           'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

This pod specification has one container that runs a bash script that writes out the value of a counter and the date once per second, and runs indefinitely. Let's create this pod in the default namespace.

```
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/counter-pod.
```

You can observe the running pod:

```
$ kubectl get pods
NAME                                           READY     STATUS    RESTARTS   AGE
counter                                        1/1       Running   0          5m
```

For a short period of time you can observe the 'Pending' pod status, because the kubelet has to download the container image first. When the pod status changes to `Running` you can use the `kubectl logs` command to view the output of this counter pod.

```
$ kubectl logs counter
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

As described in the logging overview, this command fetches log entries from the container log file. If the container is killed and then restarted by Kubernetes, you can still access logs from the previous container. However, if the pod is evicted from the node, log files are lost. Let's demonstrate this by deleting the currently running counter container:

```
$ kubectl delete pod counter
pod "counter" deleted
```

and then recreating it:

```
$ kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/counter-po
pod "counter" created
```

After some time, you can access logs from the counter pod again:

```
$ kubectl logs counter
0: Mon Jan  1 00:01:00 UTC 2001
1: Mon Jan  1 00:01:01 UTC 2001
2: Mon Jan  1 00:01:02 UTC 2001
...
```

As expected, only recent log lines are present. However, for a real-world application you will likely want to be able to access logs from all containers, especially for the debug purposes. This is exactly when the previously enabled Stackdriver Logging can help.

Viewing logs

Stackdriver Logging agent attaches metadata to each log entry, for you to use later in queries to select only the messages you're interested in: for example, the messages from a particular pod.

The most important pieces of metadata are the resource type and log name. The resource type of a container log is `container`, which is named `GKE Containers` in the UI (even if the Kubernetes cluster is not on GKE). The log name is the name of the container, so that if you have a pod with two containers, named `container_1` and `container_2` in the spec, their logs will have log names `container_1` and `container_2` respectively.

System components have resource type `compute`, which is named `GCE VM Instance` in the interface. Log names for system components are fixed. For a GKE node, every log entry from a system component has one of the following log names:

- docker
- kubelet
- kube-proxy

You can learn more about viewing logs on the dedicated Stackdriver page.

One of the possible ways to view logs is using the `gcloud logging` command line interface from the Google Cloud SDK. It uses Stackdriver Logging filtering syntax to query specific logs. For example, you can run the following command:

```
$ gcloud beta logging read 'logName="projects/$YOUR_PROJECT_ID/logs/count"' --form
...
"2: Mon Jan  1 00:01:02 UTC 2001\n"
"1: Mon Jan  1 00:01:01 UTC 2001\n"
"0: Mon Jan  1 00:01:00 UTC 2001\n"
...
"2: Mon Jan  1 00:00:02 UTC 2001\n"
"1: Mon Jan  1 00:00:01 UTC 2001\n"
"0: Mon Jan  1 00:00:00 UTC 2001\n"
```

As you can see, it outputs messages for the count container from both the first and second runs, despite the fact that the kubelet already deleted the logs for the first container.

Exporting logs

You can export logs to Google Cloud Storage or to BigQuery to run further analysis. Stackdriver Logging offers the concept of sinks, where you can specify the destination of log entries. More information is available on the Stackdriver Exporting Logs page.

Configuring Stackdriver Logging Agents

Sometimes the default installation of Stackdriver Logging may not suit your needs, for example:

- You may want to add more resources because default performance doesn't suit your needs.
- You may want to introduce additional parsing to extract more metadata from your log messages, like severity or source code reference.
- You may want to send logs not only to Stackdriver or send it to Stackdriver only partially.

In this case you need to be able to change the parameters of `DaemonSet` and `ConfigMap`.

Prerequisites

If you're using GKE and Stackdriver Logging is enabled in your cluster, you cannot change its configuration, because it's managed and supported by GKE. However, you can disable the default integration and deploy your own. Note, that you will have to support and maintain a newly deployed configuration yourself: update the image and configuration, adjust the resources and so on. To disable the default logging integration, use the following command:

```
gcloud beta container clusters update --logging-service=none CLUSTER
```

You can find notes on how to then install Stackdriver Logging agents into a running cluster in the Deploying section.

Changing **DaemonSet** parameters

When you have the Stackdriver Logging `DaemonSet` in your cluster, you can just modify the `template` field in its spec, daemonset controller will update the pods for you. For example, let’s assume you’ve just installed the Stackdriver Logging as described above. Now you want to change the memory limit to give fluentd more memory to safely process more logs.

Get the spec of `DaemonSet` running in your cluster:

```
kubectl get ds fluentd-gcp-v2.0 --namespace kube-system -o yaml > fluentd-gcp-ds.y
```

Then edit resource requirements in the spec file and update the `DaemonSet` object in the apiserver using the following command:

```
kubectl replace -f fluentd-gcp-ds.yaml
```

After some time, Stackdriver Logging agent pods will be restarted with the new configuration.

Changing fluentd parameters

Fluentd configuration is stored in the `ConfigMap` object. It is effectively a set of configuration files that are merged together. You can learn about fluentd configuration on the official site.

Imagine you want to add a new parsing logic to the configuration, so that fluentd can understand default Python logging format. An appropriate fluentd filter looks similar to this:

```
<filter reform.**>
  type parser
  format /^(?<severity>\w):(?<logger_name>\w):(?<log>.*)/
  reserve_data true
  suppress_parse_error_log true
  key_name log
</filter>
```

Now you have to put it in the configuration and make Stackdriver Logging agents pick it up. Get the current version of the Stackdriver Logging `ConfigMap` in your cluster by running the following command:

```
kubectl get cm fluentd-gcp-config --namespace kube-system -o yaml > fluentd-gcp-co
```

Then in the value for the key `containers.input.conf` insert a new filter right after the `source` section. **Note:** order is important.

Updating `ConfigMap` in the apiserver is more complicated than updating `DaemonSet`. It's better to consider `ConfigMap` to be immutable. Then, in order to update the configuration, you should create `ConfigMap` with a new name and then change `DaemonSet` to point to it using guide above.

Adding fluentd plugins

Fluentd is written in Ruby and allows to extend its capabilities using plugins. If you want to use a plugin, which is not included in the default Stackdriver Logging container image, you have to build a custom image. Imagine you want to add Kafka sink for messages from a particular container for additional processing. You can re-use the default container image sources with minor changes:

- Change Makefile to point to your container repository, e.g. `PREFIX=gcr.io/<your-project-id>`.
- Add your dependency to the Gemfile, for example `gem 'fluent-plugin-kafka'`.

Then run `make build push` from this directory. After updating `DaemonSet` to pick up the new image, you can use the plugin you installed in the fluentd configuration.

Events in Stackdriver

Kubernetes events are objects that provide insight into what is happening inside a cluster, such as what decisions were made by scheduler or why some pods were evicted from the node. You can read more about using events for debugging your application in the Application Introspection and Debugging section.

Since events are API objects, they are stored in the apiserver on master. To avoid filling up master's disk, a retention policy is enforced: events are removed one hour after the last occurrence. To provide longer history and aggregation capabilities, a third party solution should be installed to capture events.

This article describes a solution that exports Kubernetes events to Stackdriver Logging, where they can be processed and analyzed.

Note: it is not guaranteed that all events happening in a cluster will be exported to Stackdriver. One possible scenario when events will not be exported is when event exporter is not running (e.g. during restart or upgrade). In most cases it's fine to use events for purposes like setting up metrics and alerts, but you should be aware of the potential inaccuracy.

Deployment

Google Container Engine

In Google Container Engine (GKE), if cloud logging is enabled, event exporter is deployed by default to the clusters with master running version 1.7 and higher. To prevent disturbing your workloads, event exporter does not have resources set and is in the best effort QOS class, which means that it will be the first to be killed in the case of resource starvation. If you want your events to be exported, make sure you have enough resources to facilitate the event exporter pod. This may vary depending on the workload, but on average, approximately 100Mb RAM and 100m CPU is needed.

Deploying to the Existing Cluster

Deploy event exporter to your cluster using the following command:

```
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/event-export
```

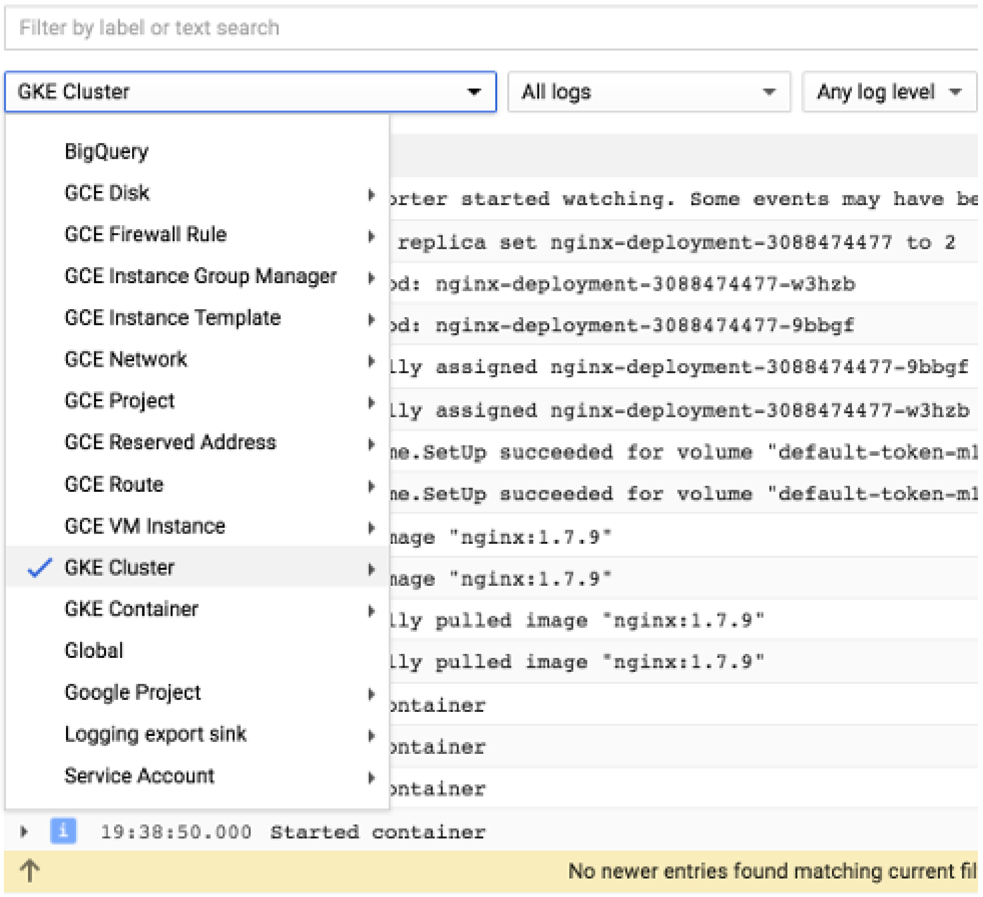
Since event exporter accesses the Kubernetes API, it requires permissions to do so. The following deployment is configured to work with RBAC authorization. It sets up a service account and a cluster role binding to allow event exporter to read events. To make sure that event exporter pod will not be evicted from the node, you can additionally set up resource requests. As mentioned earlier, 100Mb RAM and 100m CPU should be enough.

event-exporter-deploy.yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: event-exporter-sa
  namespace: default
  labels:
    app: event-exporter
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: event-exporter-rb
  namespace: default
  labels:
    app: event-exporter
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- kind: ServiceAccount
  name: event-exporter-sa
  namespace: default
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: event-exporter-v0.1.0
  namespace: default
  labels:
    app: event-exporter
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: event-exporter
    spec:
      serviceAccountName: event-exporter-sa
      containers:
      - name: event-exporter
        image: gcr.io/google-containers/event-exporter:v0.1.0
        command:
        - '/event-exporter'
      terminationGracePeriodSeconds: 30
```

User Guide

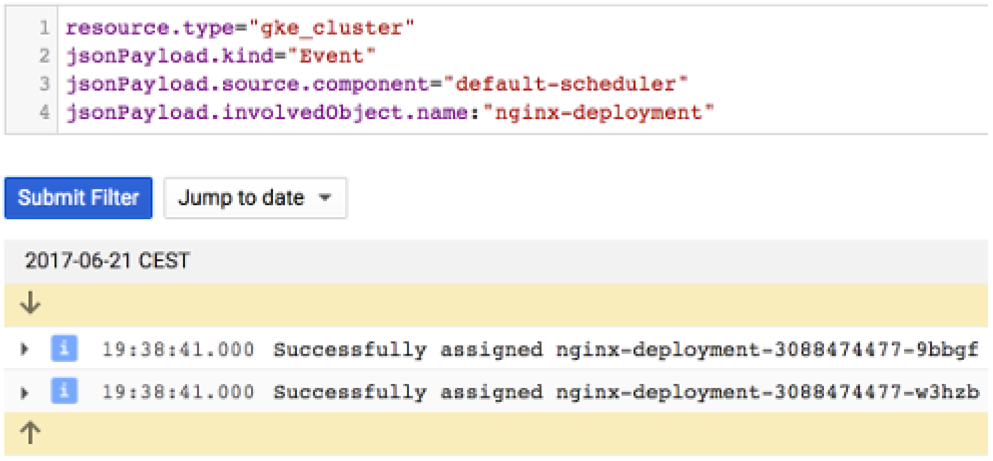
Events are exported to the `GKE Cluster` resource in Stackdriver Logging. You can find them by selecting an appropriate option from a drop-down menu of available resources:



```
orter started watching. Some events may have be
 replica set nginx-deployment-3088474477 to 2
od: nginx-deployment-3088474477-w3hzb
od: nginx-deployment-3088474477-9bbgf
lly assigned nginx-deployment-3088474477-9bbgf
lly assigned nginx-deployment-3088474477-w3hzb
ne.SetUp succeeded for volume "default-token-ml
ne.SetUp succeeded for volume "default-token-ml
nage "nginx:1.7.9"
nage "nginx:1.7.9"
lly pulled image "nginx:1.7.9"
lly pulled image "nginx:1.7.9"
ontainer
ontainer
ontainer
19:38:50.000 Started container
```

You can filter based on the event object fields using Stackdriver Logging filtering mechanism. For example, the following query will show events from the scheduler about pods from deployment `nginx-deployment`:

```
resource.type="gke_cluster"
jsonPayload.kind="Event"
jsonPayload.source.component="default-scheduler"
jsonPayload.involvedObject.name:"nginx-deployment"
```



```
resource.type="gke_cluster"
jsonPayload.kind="Event"
jsonPayload.source.component="default-scheduler"
jsonPayload.involvedObject.name:"nginx-deployment"
```

2017-06-21 CEST

```
19:38:41.000 Successfully assigned nginx-deployment-3088474477-9bbgf
19:38:41.000 Successfully assigned nginx-deployment-3088474477-w3hzb
```

Logging Using Elasticsearch and Kibana

On the Google Compute Engine (GCE) platform, the default logging support targets Stackdriver Logging, which is described in detail in the Logging With Stackdriver Logging.

This article describes how to set up a cluster to ingest logs into Elasticsearch and view them using Kibana, as an alternative to Stackdriver Logging when running on GCE. Note that Elasticsearch and Kibana cannot be setup automatically in the Kubernetes cluster hosted on Google Container Engine, you have to deploy it manually.

To use Elasticsearch and Kibana for cluster logging, you should set the following environment variable as shown below when creating your cluster with kube-up.sh:

```
KUBE_LOGGING_DESTINATION=elasticsearch
```

You should also ensure that `KUBE_ENABLE_NODE_LOGGING=true` (which is the default for the GCE platform).

Now, when you create a cluster, a message will indicate that the Fluentd log collection daemons that run on each node will target Elasticsearch:

```
$ cluster/kube-up.sh
...
Project: kubernetes-satnam
Zone: us-central1-b
... calling kube-up
Project: kubernetes-satnam
Zone: us-central1-b
+++ Staging server tars to Google Storage: gs://kubernetes-staging-e6d0e81793/deve
+++ kubernetes-server-linux-amd64.tar.gz uploaded (sha1 = 6987c098277871b6d6962314
+++ kubernetes-salt.tar.gz uploaded (sha1 = bdfc83ed6b60fa9e3bff9004b542cfc643464c
Looking for already existing resources
Starting master and configuring firewalls
Created [https://www.googleapis.com/compute/v1/projects/kubernetes-satnam/zones/us
NAME                 ZONE          SIZE_GB TYPE   STATUS
kubernetes-master-pd us-central1-b 20      pd-ssd READY
Created [https://www.googleapis.com/compute/v1/projects/kubernetes-satnam/regions/
+++ Logging using Fluentd to elasticsearch
```

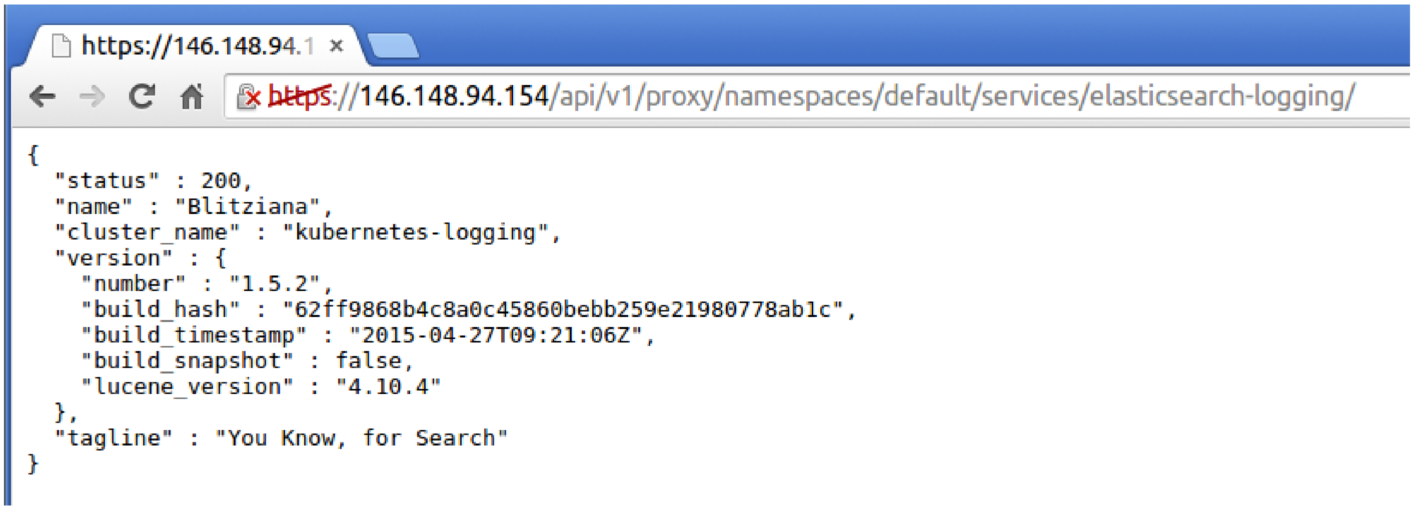
The per-node Fluentd pods, the Elasticsearch pods, and the Kibana pods should all be running in the kube-system namespace soon after the cluster comes to life.

```
$ kubectl get pods --namespace=kube-system
NAME                                           READY     STATUS    RESTARTS   AGE
elasticsearch-logging-v1-78nog                 1/1       Running   0          2h
elasticsearch-logging-v1-nj2nb                 1/1       Running   0          2h
fluentd-elasticsearch-kubernetes-node-5oq0     1/1       Running   0          2h
fluentd-elasticsearch-kubernetes-node-6896     1/1       Running   0          2h
fluentd-elasticsearch-kubernetes-node-l1ds     1/1       Running   0          2h
fluentd-elasticsearch-kubernetes-node-lz9j     1/1       Running   0          2h
kibana-logging-v1-bhpo8                        1/1       Running   0          2h
kube-dns-v3-7r1l9                              3/3       Running   0          2h
monitoring-heapster-v4-yl332                   1/1       Running   1          2h
monitoring-influx-grafana-v1-o79xf             2/2       Running   0          2h
```

The `fluentd-elasticsearch` pods gather logs from each node and send them to the `elasticsearch-logging` pods, which are part of a service named `elasticsearch-logging`. These Elasticsearch pods store the logs and expose them via a REST API. The `kibana-logging` pod provides a web UI for reading the logs stored in Elasticsearch, and is part of a service named `kibana-logging`.

The Elasticsearch and Kibana services are both in the `kube-system` namespace and are not directly exposed via a publicly reachable IP address. To reach them, follow the instructions for Accessing services running in a cluster.

If you try accessing the `elasticsearch-logging` service in your browser, you'll see a status page that looks something like this:

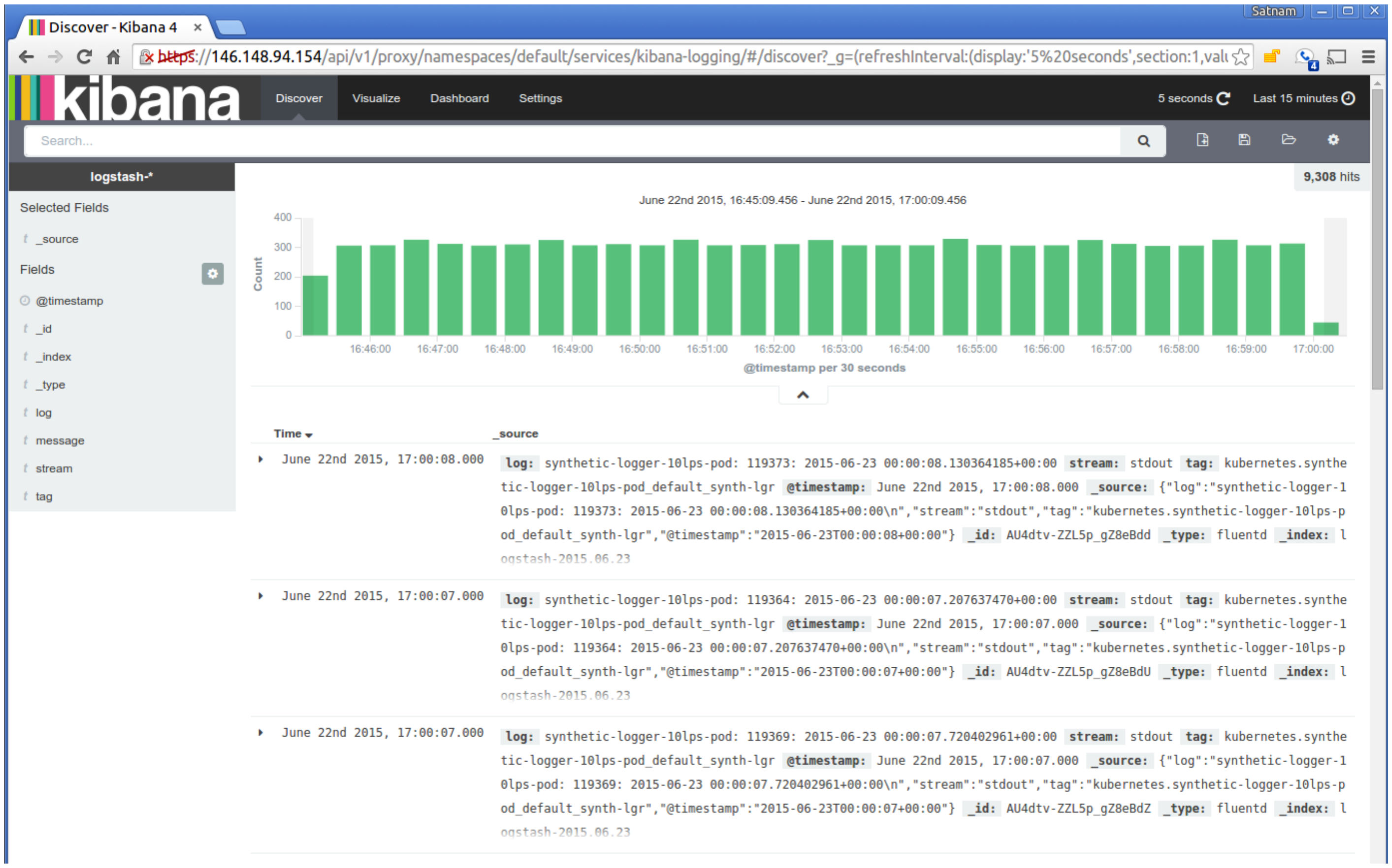

https://146.148.94.1
~~https~~://146.148.94.154/api/v1/proxy/namespaces/default/services/elasticsearch-logging/

```
{
  "status" : 200,
  "name" : "Blitziana",
  "cluster_name" : "kubernetes-logging",
  "version" : {
    "number" : "1.5.2",
    "build_hash" : "62ff9868b4c8a0c45860bebb259e21980778ab1c",
    "build_timestamp" : "2015-04-27T09:21:06Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}
```

You can now type Elasticsearch queries directly into the browser, if you'd like. See Elasticsearch's documentation for more details on how to do so.

Alternatively, you can view your cluster's logs using Kibana (again using the instructions for accessing a service running in the cluster). The first time you visit the Kibana URL you will be presented with a page that asks you to configure your view of the ingested logs. Select the option for timeseries values and select `@timestamp` . On the following page select the `Discover` tab and then you should be able to see the ingested logs. You can set the refresh interval to 5 seconds to have the logs regularly refreshed.

Here is a typical view of ingested logs from the Kibana viewer:



```
Time ▾                      _source
▸ June 22nd 2015, 17:00:08.000   log: synthetic-logger-10lps-pod: 119373: 2015-06-23 00:00:08.130364185+00:00 stream: stdout tag: kubernetes.synthe
                                 tic-logger-10lps-pod_default_synth-lgr @timestamp: June 22nd 2015, 17:00:08.000 _source: {"log":"synthetic-logger-1
                                 0lps-pod: 119373: 2015-06-23 00:00:08.130364185+00:00\n","stream":"stdout","tag":"kubernetes.synthetic-logger-10lps-p
                                 od_default_synth-lgr","@timestamp":"2015-06-23T00:00:08+00:00"} _id: AU4dtv-ZZL5p_gZ8eBdd _type: fluentd _index: l
                                 ogstash-2015.06.23

▸ June 22nd 2015, 17:00:07.000   log: synthetic-logger-10lps-pod: 119364: 2015-06-23 00:00:07.207637470+00:00 stream: stdout tag: kubernetes.synthe
                                 tic-logger-10lps-pod_default_synth-lgr @timestamp: June 22nd 2015, 17:00:07.000 _source: {"log":"synthetic-logger-1
                                 0lps-pod: 119364: 2015-06-23 00:00:07.207637470+00:00\n","stream":"stdout","tag":"kubernetes.synthetic-logger-10lps-p
                                 od_default_synth-lgr","@timestamp":"2015-06-23T00:00:07+00:00"} _id: AU4dtv-ZZL5p_gZ8eBdU _type: fluentd _index: l
                                 ogstash-2015.06.23

▸ June 22nd 2015, 17:00:07.000   log: synthetic-logger-10lps-pod: 119369: 2015-06-23 00:00:07.720402961+00:00 stream: stdout tag: kubernetes.synthe
                                 tic-logger-10lps-pod_default_synth-lgr @timestamp: June 22nd 2015, 17:00:07.000 _source: {"log":"synthetic-logger-1
                                 0lps-pod: 119369: 2015-06-23 00:00:07.720402961+00:00\n","stream":"stdout","tag":"kubernetes.synthetic-logger-10lps-p
                                 od_default_synth-lgr","@timestamp":"2015-06-23T00:00:07+00:00"} _id: AU4dtv-ZZL5p_gZ8eBdZ _type: fluentd _index: l
                                 ogstash-2015.06.23
```

Kibana opens up all sorts of powerful options for exploring your logs! For some ideas on how to dig into it, check out Kibana's documentation.

Determine the Reason for Pod Failure

This page shows how to write and read a Container termination message.

Termination messages provide a way for containers to write information about fatal events to a location where it can be easily retrieved and surfaced by tools like dashboards and monitoring software. In most cases, information that you put in a termination message should also be written to the general Kubernetes logs.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Writing and reading a termination message

In this exercise, you create a Pod that runs one container. The configuration file specifies a command that runs when the container starts.

termination.yaml

termination.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
  - name: termination-demo-container
    image: debian
    command: ["/bin/sh"]
    args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

1. Create a Pod based on the YAML configuration file:

   ```
   kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/terminat
   ```

   In the YAML file, in the `cmd` and `args` fields, you can see that the container sleeps for 10 seconds and then writes "Sleep expired" to the `/dev/termination-log` file. After the container writes the "Sleep expired" message, it terminates.

2. Display information about the Pod:

   ```
   kubectl get pod termination-demo
   ```

   Repeat the preceding command until the Pod is no longer running.

3. Display detailed information about the Pod:

   ```
   kubectl get pod --output=yaml
   ```

   The output includes the "Sleep expired" message:

```
apiVersion: v1
kind: Pod
...
    lastState:
      terminated:
        containerID: ...
        exitCode: 0
        finishedAt: ...
        message: |
          Sleep expired
        ...
```

4. Use a Go template to filter the output so that it includes only the termination message:

```
kubectl get pod termination-demo -o go-template="{{range .status.containerStatus
```

Setting the termination log file

By default Kubernetes retrieves termination messages from `/dev/termination-log`. To change this to a different file, specify a `terminationMessagePath` field for your Container.

For example, suppose your Container writes termination messages to `/tmp/my-log`, and you want Kubernetes to retrieve those messages. Set `terminationMessagePath` as shown here:

```
apiVersion: v1
kind: Pod
metadata:
  name: msg-path-demo
spec:
  containers:
  - name: msg-path-demo-container
    image: debian
    terminationMessagePath: "/tmp/my-log"
```

What’s next

- See the `terminationMessagePath` field in Container.
- Learn about retrieving logs.
- Learn about Go templates.

Debug Init Containers

This page shows how to investigate problems related to the execution of Init Containers. The example command lines below refer to the Pod as `<pod-name>` and the Init Containers as `<init-container-1>` and `<init-container-2>`.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes
- You should be familiar with the basics of Init Containers.
- You should have Configured an Init Container.

Checking the status of Init Containers

Display the status of your pod:

```
kubectl get pod <pod-name>
```

For example, a status of `Init:1/2` indicates that one of two Init Containers has completed successfully:

```
NAME         READY     STATUS     RESTARTS   AGE
<pod-name>   0/1       Init:1/2   0          7s
```

See [Understanding Pod status]() for more examples of status values and their meanings.

Getting details about Init Containers

View more detailed information about Init Container execution:

```
kubectl describe pod <pod-name>
```

For example, a Pod with two Init Containers might show the following:

```
Init Containers:
  <init-container-1>:
    Container ID:    ...
    ...
    State:           Terminated
      Reason:        Completed
      Exit Code:     0
      Started:       ...
      Finished:      ...
    Ready:           True
    Restart Count:   0
    ...
  <init-container-2>:
    Container ID:    ...
    ...
    State:           Waiting
      Reason:        CrashLoopBackOff
    Last State:      Terminated
      Reason:        Error
      Exit Code:     1
      Started:       ...
      Finished:      ...
    Ready:           False
    Restart Count:   3
    ...
```

You can also access the Init Container statuses programmatically by reading the `status.initContainerStatuses` field on the Pod Spec:

```
kubectl get pod nginx --template '{{.status.initContainerStatuses}}'
```

This command will return the same information as above in raw JSON.

Accessing logs from Init Containers

Pass the Init Container name along with the Pod name to access its logs.

```
kubectl logs <pod-name> -c <init-container-2>
```

Init Containers that run a shell script print commands as they're executed. For example, you can do this in Bash by running `set -x` at the beginning of the script.

Understanding Pod status

A Pod status beginning with `Init:` summarizes the status of Init Container execution. The table below describes some example status values that you might see while debugging Init Containers.

Status	Meaning
`Init:N/M`	The Pod has `M` Init Containers, and `N` have completed so far.
`Init:Error`	An Init Container has failed to execute.
`Init:CrashLoopBackOff`	An Init Container has failed repeatedly.
`Pending`	The Pod has not yet begun executing Init Containers.
`PodInitializing` or `Running`	The Pod has already finished executing Init Containers.

Debug Pods and Replication Controllers

Debugging pods

The first step in debugging a pod is taking a look at it. Check the current state of the pod and recent events with the following command:

```
$ kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all `Running`? Have there been recent restarts?

Continue debugging depending on the state of the pods.

My pod stays pending

If a pod is stuck in `Pending` it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kubectl describe ...` command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

Insufficient resources

You may have exhausted the supply of CPU or Memory in your cluster. In this case you can try several things:

- Add more nodes to the cluster.
- Terminate unneeded pods to make room for pending pods.
- Check that the pod is not larger than your nodes. For example, if all nodes have a capacity of `cpu:1`, then a pod with a request of `cpu: 1.1` will never be scheduled.

  You can check node capacities with the `kubectl get nodes -o <format>` command. Here are some example command lines that extract just the necessary information:

  ```
  kubectl get nodes -o yaml | grep '\sname\|cpu\|memory'
  kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, cap: .status
  ```

  The resource quota feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

Using hostPort

When you bind a pod to a `hostPort` there are a limited number of places that the pod can be scheduled. In most cases, `hostPort` is unnecessary; try using a service object to expose your pod. If you do require `hostPort` then you can only schedule as many pods as there are nodes in your container cluster.

My pod stays waiting

If a pod is stuck in the `Waiting` state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of `Waiting` pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

My pod is crashing or otherwise unhealthy

First, take a look at the logs of the current container:

```
$ kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
$ kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Alternately, you can run commands inside that container with `exec`:

```
$ kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${AR
```

Note that `-c ${CONTAINER_NAME}` is optional and can be omitted for pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run:

```
$ kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

If none of these approaches work, you can find the host machine that the pod is running on and SSH into that host.

Debugging Replication Controllers

Replication controllers are fairly straightforward. They can either create pods or they can't. If they can't create pods, then please refer to the instructions above to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to inspect events related to the replication controller.

Debug Services

An issue that comes up rather frequently for new installations of Kubernetes is that `Services` are not working properly. You've run all your `Pods` and `Deployments`, but you get no response when you try to access them. This document will hopefully help you to figure out what's going wrong.

Conventions

Throughout this doc you will see various commands that you can run. Some commands need to be run within a `Pod`, others on a Kubernetes `Node`, and others can run anywhere you have `kubectl` and credentials for the cluster. To make it clear what is expected, this document will use the following conventions.

If the command "COMMAND" is expected to run in a `Pod` and produce "OUTPUT":

```
u@pod$ COMMAND
OUTPUT
```

If the command "COMMAND" is expected to run on a `Node` and produce "OUTPUT":

```
u@node$ COMMAND
OUTPUT
```

If the command is "kubectl ARGS":

```
$ kubectl ARGS
OUTPUT
```

Running commands in a Pod

For many steps here you will want to see what a `Pod` running in the cluster sees. You can start a busybox `Pod` and run commands in it:

```
$ kubectl run -i --tty busybox --image=busybox --generator="run-pod/v1"
Waiting for pod default/busybox to be running, status is Pending, pod ready: false

Hit enter for command prompt

/ #
```

If you already have a running `Pod`, run a command in it using:

```
$ kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

or run an interactive shell with:

```
$ kubectl exec -ti <POD-NAME> -c <CONTAINER-NAME> sh
/ #
```

Setup

For the purposes of this walk-through, let's run some `Pods`. Since you're probably debugging your own `Service` you can substitute your own details, or you can follow along and get a second data point.

```
$ kubectl run hostnames --image=gcr.io/google_containers/serve_hostname \
                        --labels=app=hostnames \
                        --port=9376 \
                        --replicas=3
deployment "hostnames" created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands. Note that this is the same as if you had started the `Deployment` with the following YAML:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hostnames
spec:
  selector:
    app: hostnames
  replicas: 3
  template:
    metadata:
      labels:
        app: hostnames
    spec:
      containers:
      - name: hostnames
        image: gcr.io/google_containers/serve_hostname
        ports:
        - containerPort: 9376
          protocol: TCP
```

Confirm your `Pods` are running:

```
$ kubectl get pods -l app=hostnames
NAME                        READY     STATUS    RESTARTS   AGE
hostnames-632524106-bbpiw   1/1       Running   0          2m
hostnames-632524106-ly40y   1/1       Running   0          2m
hostnames-632524106-tlaok   1/1       Running   0          2m
```

Does the Service exist?

The astute reader will have noticed that we did not actually create a `Service` yet - that is intentional. This is a step that sometimes gets forgotten, and is the first thing to check.

So what would happen if I tried to access a non-existent `Service`? Assuming you have another `Pod` that consumes this `Service` by name you would get something like:

```
u@pod$ wget -qO- hostnames
wget: bad address 'hostname'
```

or:

```
u@pod$ echo $HOSTNAMES_SERVICE_HOST
```

So the first thing to check is whether that `Service` actually exists:

```
$ kubectl get svc hostnames
Error from server (NotFound): services "hostnames" not found
```

So we have a culprit, let's create the `Service`. As before, this is for the walk-through - you can use your own `Service`'s details here.

```
$ kubectl expose deployment hostnames --port=80 --target-port=9376
service "hostnames" exposed
```

And read it back, just to be sure:

```
$ kubectl get svc hostnames
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
hostnames   10.0.0.226   <none>        80/TCP    5s
```

As before, this is the same as if you had started the `Service` with YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: hostnames
spec:
  selector:
    app: hostnames
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 9376
```

Now you can confirm that the `Service` exists.

Does the Service work by DNS?

From a `Pod` in the same `Namespace`:

```
u@pod$ nslookup hostnames
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames
Address: 10.0.1.175
```

If this fails, perhaps your `Pod` and `Service` are in different `Namespaces`, try a namespace-qualified name:

```
u@pod$ nslookup hostnames.default
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames.default
Address: 10.0.1.175
```

If this works, you'll need to ensure that `Pods` and `Services` run in the same `Namespace`. If this still fails, try a fully-qualified name:

```
u@pod$ nslookup hostnames.default.svc.cluster.local
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames.default.svc.cluster.local
Address: 10.0.1.175
```

Note the suffix here: "default.svc.cluster.local". The "default" is the `Namespace` we're operating in. The "svc" denotes that this is a `Service`. The "cluster.local" is your cluster domain.

You can also try this from a `Node` in the cluster (note: 10.0.0.10 is my DNS `Service`):

```
u@node$ nslookup hostnames.default.svc.cluster.local 10.0.0.10
Server:         10.0.0.10
Address:        10.0.0.10#53

Name:   hostnames.default.svc.cluster.local
Address: 10.0.1.175
```

If you are able to do a fully-qualified name lookup but not a relative one, you need to check that your `kubelet` is running with the right flags. The `--cluster-dns` flag needs to point to your DNS `Service`'s IP and the `--cluster-domain` flag needs to be your cluster's domain - we assumed "cluster.local" in this document, but yours might be different, in which case you should change that in all of the commands above.

Does any Service exist in DNS?

If the above still fails - DNS lookups are not working for your `Service` - we can take a step back and see what else is not working. The Kubernetes master `Service` should always work:

```
u@pod$ nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes
Address 1: 10.0.0.1
```

If this fails, you might need to go to the kube-proxy section of this doc, or even go back to the top of this document and start over, but instead of debugging your own `Service`, debug DNS.

Does the Service work by IP?

The next thing to test is whether your `Service` works at all. From a `Node` in your cluster, access the `Service`'s IP (from `kubectl get` above).

```
u@node$ curl 10.0.1.175:80
hostnames-0uton

u@node$ curl 10.0.1.175:80
hostnames-yp2kp

u@node$ curl 10.0.1.175:80
hostnames-bvc05
```

If your `Service` is working, you should get correct responses. If not, there are a number of things that could be going wrong. Read on.

Is the Service correct?

It might sound silly, but you should really double and triple check that your `Service` is correct and matches your `Pods`. Read back your `Service` and verify it:

```
$ kubectl get service hostnames -o json
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "hostnames",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/services/hostnames",
        "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
        "resourceVersion": "347189",
        "creationTimestamp": "2015-07-07T15:24:29Z",
        "labels": {
            "app": "hostnames"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "default",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376,
                "nodePort": 0
            }
        ],
        "selector": {
            "app": "hostnames"
        },
        "clusterIP": "10.0.1.175",
        "type": "ClusterIP",
        "sessionAffinity": "None"
    },
    "status": {
        "loadBalancer": {}
    }
}
```

Is the port you are trying to access in `spec.ports[]`? Is the `targetPort` correct for your `Pods`? If you meant it to be a numeric port, is it a number (9376) or a string "9376"? If you meant it to be a named port, do your `Pods` expose a port with the same name? Is the port's `protocol` the same as the `Pod`'s?

Does the Service have any Endpoints?

If you got this far, we assume that you have confirmed that your `Service` exists and is resolved by DNS. Now let's check that the `Pods` you ran are actually being selected by the `Service`.

Earlier we saw that the `Pods` were running. We can re-check that:

```
$ kubectl get pods -l app=hostnames
NAME              READY     STATUS    RESTARTS   AGE
hostnames-0uton   1/1       Running   0          1h
hostnames-bvc05   1/1       Running   0          1h
hostnames-yp2kp   1/1       Running   0          1h
```

The "AGE" column says that these `Pods` are about an hour old, which implies that they are running fine and not crashing.

The `-l app=hostnames` argument is a label selector - just like our `Service` has. Inside the Kubernetes system is a control loop which evaluates the selector of every `Service` and saves the results into an `Endpoints` object.

```
$ kubectl get endpoints hostnames
NAME        ENDPOINTS
hostnames   10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376
```

This confirms that the control loop has found the correct `Pods` for your `Service`. If the `hostnames` row is blank, you should check that the `spec.selector` field of your `Service` actually selects for `metadata.labels` values on your `Pods`.

Are the Pods working?

At this point, we know that your `Service` exists and has selected your `Pods`. Let's check that the `Pods` are actually working - we can bypass the `Service` mechanism and go straight to the `Pods`.

```
u@pod$ wget -qO- 10.244.0.5:9376
hostnames-0uton

pod $ wget -qO- 10.244.0.6:9376
hostnames-bvc05

u@pod$ wget -qO- 10.244.0.7:9376
hostnames-yp2kp
```

We expect each `Pod` in the `Endpoints` list to return its own hostname. If this is not what happens (or whatever the correct behavior is for your own `Pods`), you should investigate what's happening there. You might find `kubectl logs` to be useful or `kubectl exec` directly to your `Pods` and check service from there.

Is the kube-proxy working?

If you get here, your `Service` is running, has `Endpoints`, and your `Pods` are actually serving. At this point, the whole `Service` proxy mechanism is suspect. Let's confirm it, piece by piece.

Is kube-proxy running?

Confirm that `kube-proxy` is running on your `Nodes`. You should get something like the below:

```
u@node$ ps auxw | grep kube-proxy
root  4194  0.4  0.1 101864 17696 ?    Sl Jul04  25:43 /usr/local/bin/kube-proxy -
```

Next, confirm that it is not failing something obvious, like contacting the master. To do this, you'll have to look at the logs. Accessing the logs depends on your `Node` OS. On some OSes it is a file, such as /var/log/kube-proxy.log, while other OSes use `journalctl` to access logs. You should see something like:

```
I1027 22:14:53.995134    5063 server.go:200] Running in resource-only container "/
I1027 22:14:53.998163    5063 server.go:247] Using iptables Proxier.
I1027 22:14:53.999055    5063 server.go:255] Tearing down userspace rules. Errors
I1027 22:14:54.038140    5063 proxier.go:352] Setting endpoints for "kube-system/k
I1027 22:14:54.038164    5063 proxier.go:352] Setting endpoints for "kube-system/k
I1027 22:14:54.038209    5063 proxier.go:352] Setting endpoints for "default/kuber
I1027 22:14:54.038238    5063 proxier.go:429] Not syncing iptables until Services
I1027 22:14:54.040048    5063 proxier.go:294] Adding new service "default/kubernet
I1027 22:14:54.040154    5063 proxier.go:294] Adding new service "kube-system/kube
I1027 22:14:54.040223    5063 proxier.go:294] Adding new service "kube-system/kube
```

If you see error messages about not being able to contact the master, you should double-check your `Node` configuration and installation steps.

Is kube-proxy writing iptables rules?

One of the main responsibilities of `kube-proxy` is to write the `iptables` rules which implement `Services`. Let's check that those rules are getting written.

The kube-proxy can run in either "userspace" mode or "iptables" mode. Hopefully you are using the newer, faster, more stable "iptables" mode. You should see one of the following cases.

Userspace

```
u@node$ iptables-save | grep hostnames
-A KUBE-PORTALS-CONTAINER -d 10.0.1.175/32 -p tcp -m comment --comment "default/ho
-A KUBE-PORTALS-HOST -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnam
```

There should be 2 rules for each port on your `Service` (just one in this example) - a "KUBE-PORTALS-CONTAINER" and a "KUBE-PORTALS-HOST". If you do not see these, try restarting `kube-proxy` with the `-V` flag set to 4, and then look at the logs again.

Iptables

```
u@node$ iptables-save | grep hostnames
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --comment "default/hostna
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m t
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "default/hostna
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostnames:" -m t
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostna
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m t
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames:
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statisti
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -m statisti
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/hostnames:" -j KUBE-SEP
```

There should be 1 rule in `KUBE-SERVICES`, 1 or 2 rules per endpoint in `KUBE-SVC-(hash)` (depending on `SessionAffinity`), one `KUBE-SEP-(hash)` chain per endpoint, and a few rules in each `KUBE-SEP-(hash)` chain. The exact rules will vary based on your exact config (including node-ports and load-balancers).

Is kube-proxy proxying?

Assuming you do see the above rules, try again to access your `Service` by IP:

```
u@node$ curl 10.0.1.175:80
hostnames-0uton
```

If this fails and you are using the userspace proxy, you can try accessing the proxy directly. If you are using the iptables proxy, skip this section.

Look back at the `iptables-save` output above, and extract the port number that `kube-proxy` is using for your `Service`. In the above examples it is “48577”. Now connect to that:

```
u@node$ curl localhost:48577
hostnames-yp2kp
```

If this still fails, look at the `kube-proxy` logs for specific lines like:

```
Setting endpoints for default/hostnames:default to [10.244.0.5:9376 10.244.0.6:937
```

If you don't see those, try restarting `kube-proxy` with the `-V` flag set to 4, and then look at the logs again.

Services provide load balancing across a set of pods. There are several common problems that can make services not work properly. The following instructions should help debug service problems.

First, verify that there are endpoints for the service. For every service object, the apiserver makes an `endpoints` resource available.

You can view this resource with:

```
$ kubectl get endpoints ${SERVICE_NAME}
```

Make sure that the endpoints match up with the number of containers that you expect to be a member of your service. For example, if your service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the service's endpoints.

My service is missing endpoints

If you are missing endpoints, try listing pods using the labels that service uses. Imagine that you have a service where the labels are:

```
...
spec:
  - selector:
     name: nginx
     type: frontend
```

You can use:

```
$ kubectl get pods --selector=name=nginx,type=frontend
```

to list pods that match this selector. Verify that the list matches the pods that you expect to provide your service.

If the list of pods matches expectations, but your endpoints are still empty, it's possible that you don't have the right ports exposed. If your service has a `containerPort` specified, but the pods that are selected don't have that port listed, then they won't be added to the endpoints list.

Verify that the pod's `containerPort` matches up with the service's `containerPort`.

Network traffic is not forwarded

If you can connect to the service, but the connection is immediately dropped, and there are endpoints in the endpoints list, it's likely that the proxy can't contact your pods.

There are three things to check:

- Are your pods working correctly? Look for restart count, and debug pods.
- Can you connect to your pods directly? Get the IP address for the pod, and try to connect directly to that IP.
- Is your application serving on the port that you configured? Container Engine doesn't do port remapping, so if your application serves on 8080, the `containerPort` field needs to be 8080.

A Pod cannot reach itself via Service IP

This mostly happens when `kube-proxy` is running in `iptables` mode and Pods are connected with bridge network. The `Kubelet` exposes a `hairpin-mode` flag that allows endpoints of a Service to loadbalance back to themselves if they try to access their own Service VIP. The `hairpin-mode` flag must either be set to `hairpin-veth` or `promiscuous-bridge`.

The common steps to trouble shoot this are as follows:

- Confirm `hairpin-mode` is set to `hairpin-veth` or `promiscuous-bridge`. You should see something like the below. `hairpin-mode` is set to `promiscuous-bridge` in the following example.

```
u@node$ ps auxw|grep kubelet
root      3392  1.1  0.8 186804 65208 ?        Sl   00:51  11:11 /usr/local/bin/ku
```

- Confirm the effective `hairpin-mode`. To do this, you'll have to look at kubelet log. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kubelet.log, while other OSes use `journalctl` to access logs. Please be noted that the effective hairpin mode may not match `--hairpin-mode` flag due to compatibility. Check if there is any log lines with key word `hairpin` in kubelet.log. There should be log lines indicating the effective hairpin mode, like something below.

```
I0629 00:51:43.648698    3252 kubelet.go:380] Hairpin mode set to "promiscuous-bri
```

- If the effective hairpin mode is `hairpin-veth`, ensure the `Kubelet` has the permission to operate in `/sys` on node. If everything works properly, you should see something like:

```
u@node$ for intf in /sys/devices/virtual/net/cbr0/brif/*; do cat $intf/hairpin_mod
1
1
1
1
```

- If the effective hairpin mode is `promiscuous-bridge`, ensure `Kubelet` has the permission to manipulate linux bridge on node. If cbr0` bridge is used and configured properly, you should see:

```
u@node$ ifconfig cbr0 |grep PROMISC
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```

- Seek help if none of above works out.

Seek help

If you get this far, something very strange is happening. Your `Service` is running, has `Endpoints`, and your `Pods` are actually serving. You have DNS working, `iptables` rules installed, and `kube-proxy` does not seem to be misbehaving. And yet your `Service` is not working. You should probably let us know, so we can help investigate!

Contact us on Slack or email or GitHub.

More information

Visit troubleshooting document for more information.

Troubleshoot Clusters

This doc is about cluster troubleshooting; we assume you have already ruled out your application as the root cause of the problem you are experiencing. See the application troubleshooting guide for tips on application debugging. You may also visit troubleshooting document for more information.

Listing your cluster

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run

```
kubectl get nodes
```

And verify that all of the nodes you expect to see are present and that they are all in the `Ready` state.

Looking at logs

For now, digging deeper into the cluster requires logging into the relevant machines. Here are the locations of the relevant log files. (note that on systemd-based systems, you may need to use `journalctl` instead)

Master

- /var/log/kube-apiserver.log - API Server, responsible for serving the API
- /var/log/kube-scheduler.log - Scheduler, responsible for making scheduling decisions
- /var/log/kube-controller-manager.log - Controller that manages replication controllers

Worker Nodes

- /var/log/kubelet.log - Kubelet, responsible for running containers on the node
- /var/log/kube-proxy.log - Kube Proxy, responsible for service load balancing

A general overview of cluster failure modes

This is an incomplete list of things that could go wrong, and how to adjust your cluster setup to mitigate the problems.

Root causes:

- VM(s) shutdown
- Network partition within cluster, or between cluster and users
- Crashes in Kubernetes software
- Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
- Operator error, e.g. misconfigured Kubernetes software or application software

Specific scenarios:

- Apiserver VM shutdown or apiserver crashing
    - Results
        - unable to stop, update, or start new pods, services, replication controller
        - existing pods and services should continue to work normally, unless they depend on the Kubernetes API
- Apiserver backing storage lost
    - Results
        - apiserver should fail to come up
        - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
        - manual recovery or recreation of apiserver state necessary before apiserver is restarted

- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes
  - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
  - in future, these will be replicated as well and may not be co-located
  - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
  - Results
    - pods on that Node stop running
- Network partition
  - Results
    - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
  - Results
    - crashing kubelet cannot start new pods on the node
    - kubelet might delete the pods or not
    - node marked unhealthy
    - replication controllers start new pods elsewhere
- Cluster operator error
  - Results
    - loss of pods, services, etc
    - lost of apiserver backing store
    - users unable to read API
    - etc.

Mitigations:

- Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
    - Mitigates: Apiserver VM shutdown or apiserver crashing
    - Mitigates: Supporting services VM shutdown or crashes
- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
    - Mitigates: Apiserver backing storage lost
- Action: Use (experimental) high-availability configuration
    - Mitigates: Master VM shutdown or master components (scheduler, API server, controller-managing) crashing
        - Will tolerate one or more simultaneous node or component failures
    - Mitigates: Apiserver backing storage (i.e., etcd's data directory) lost
        - Assuming you used clustered etcd.
- Action: Snapshot apiserver PDs/EBS-volumes periodically
    - Mitigates: Apiserver backing storage lost
    - Mitigates: Some cases of operator error
    - Mitigates: Some cases of Kubernetes software fault
- Action: use replication controller and services in front of pods
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault
- Action: applications (containers) designed to tolerate unexpected restarts
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault
- Action: Multiple independent clusters (and avoid making risky changes to all clusters at once)
    - Mitigates: Everything listed above.

Troubleshoot Applications

This guide is to help users debug applications that are deployed into Kubernetes and not behaving correctly. This is *not* a guide for people who want to debug their cluster. For that you should check out this guide.

Diagnosing the problem

The first step in troubleshooting is triage. What is the problem? Is it your Pods, your Replication Controller or your Service?

- Debugging Pods
- Debugging Replication Controllers
- Debugging Services

Debugging Pods

The first step in debugging a Pod is taking a look at it. Check the current state of the Pod and recent events with the following command:

```
$ kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all `Running`? Have there been recent restarts?

Continue debugging depending on the state of the pods.

My pod stays pending

If a Pod is stuck in `Pending` it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kubectl describe ...` command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

- **You don't have enough resources**: You may have exhausted the supply of CPU or Memory in your cluster, in this case you need to delete Pods, adjust resource requests, or add new nodes to your cluster. See Compute Resources document for more information.
- **You are using `hostPort`**: When you bind a Pod to a `hostPort` there are a limited number of places that pod can be scheduled. In most cases, `hostPort` is unnecessary, try using a Service object to expose your Pod. If you do require `hostPort` then you can only schedule as many Pods as there are nodes in your Kubernetes cluster.

My pod stays waiting

If a Pod is stuck in the `Waiting` state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of `Waiting` pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

My pod is crashing or otherwise unhealthy

First, take a look at the logs of the current container:

```
$ kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
$ kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Alternately, you can run commands inside that container with `exec` :

```
$ kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${AR
```

Note that `-c ${CONTAINER_NAME}` is optional and can be omitted for Pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run

```
$ kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

If none of these approaches work, you can find the host machine that the pod is running on and SSH into that host, but this should generally not be necessary given tools in the Kubernetes API. Therefore, if you find yourself needing to ssh into a machine, please file a feature request on GitHub describing your use case and why these tools are insufficient.

My pod is running but not doing what I told it to do

If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. `mypod.yaml` file on your local machine), and that the error was silently ignored when you created the pod. Often a section of the pod description is nested incorrectly, or a key name is typed incorrectly, and so the key is ignored. For example, if you misspelled `command` as `commnd` then the pod will be created but will not use the command line you intended it to use.

The first thing to do is to delete your pod and try creating it again with the `--validate` option. For example, run `kubectl create --validate -f mypod.yaml` . If you misspelled `command` as `commnd` then will give an error like this:

```
I0805 10:43:25.129850   46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973   46757 schema.go:129] this may be a false alarm, see https:
pods/mypod
```

The next thing to check is whether the pod on the apiserver matches the pod you meant to create (e.g. in a yaml file on your local machine). For example, run `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml` and then manually compare the original pod description, `mypod.yaml` with the one you got back from apiserver, `mypod-on-apiserver.yaml`. There will typically be some lines on the "apiserver" version that are not on the original version. This is expected. However, if there are lines on the original that are not on the apiserver version, then this may indicate a problem with your pod spec.

Debugging Replication Controllers

Replication controllers are fairly straightforward. They can either create Pods or they can't. If they can't create pods, then please refer to the instructions above to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to introspect events related to the replication controller.

Debugging Services

Services provide load balancing across a set of pods. There are several common problems that can make Services not work properly. The following instructions should help debug Service problems.

First, verify that there are endpoints for the service. For every Service object, the apiserver makes an `endpoints` resource available.

You can view this resource with:

```
$ kubectl get endpoints ${SERVICE_NAME}
```

Make sure that the endpoints match up with the number of containers that you expect to be a member of your service. For example, if your Service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the Service's endpoints.

My service is missing endpoints

If you are missing endpoints, try listing pods using the labels that Service uses. Imagine that you have a Service where the labels are:

```
...
spec:
  - selector:
     name: nginx
     type: frontend
```

You can use:

```
$ kubectl get pods --selector=name=nginx,type=frontend
```

to list pods that match this selector. Verify that the list matches the Pods that you expect to provide your Service.

If the list of pods matches expectations, but your endpoints are still empty, it's possible that you don't have the right ports exposed. If your service has a `containerPort` specified, but the Pods that are selected don't have that port listed, then they won't be added to the endpoints list.

Verify that the pod's `containerPort` matches up with the Service's `containerPort`

Network traffic is not forwarded

If you can connect to the service, but the connection is immediately dropped, and there are endpoints in the endpoints list, it's likely that the proxy can't contact your pods.

There are three things to check:

- Are your pods working correctly? Look for restart count, and debug pods.
- Can you connect to your pods directly? Get the IP address for the Pod, and try to connect directly to that IP.
- Is your application serving on the port that you configured? Kubernetes doesn't do port remapping, so if your application serves on 8080, the `containerPort` field needs to be 8080.

More information

If none of the above solves your problem, follow the instructions in Debugging Service document to make sure that your `Service` is running, has `Endpoints`, and your `Pods` are actually serving; you have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving.

You may also visit troubleshooting document for more information.

Debug a StatefulSet

This task shows you how to debug a StatefulSet.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster.
- You should have a StatefulSet running that you want to investigate.

Debugging a StatefulSet

In order to list all the pods which belong to a StatefulSet, which have a label `app=myapp` set on them, you can use the following:

```
kubectl get pods -l app=myapp
```

If you find that any Pods listed are in `Unknown` or `Terminating` state for an extended period of time, refer to the Deleting StatefulSet Pods task for instructions on how to deal with them. You can debug individual Pods in a StatefulSet using the Debugging Pods guide.

What's next

Learn more about debugging an init-container.

Application Introspection and Debugging

Once your application is running, you'll inevitably need to debug problems with it. Earlier we described how you can use `kubectl get pods` to retrieve simple status information about your pods. But there are a number of ways to get even more information about your application.

Using `kubectl describe pod` to fetch details about pods

For this example we'll use a Deployment to create two pods, similar to the earlier example.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

Copy this to a file *./my-nginx-dep.yaml*

```
$ kubectl create -f ./my-nginx-dep.yaml
deployment "nginx-deployment" created
```

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1       Running   0          11s
nginx-deployment-1006230814-fmgu3   1/1       Running   0          11s
```

We can retrieve a lot more information about each of these pods using `kubectl describe pod`. For example:

```
$ kubectl describe pod nginx-deployment-1006230814-6winp
Name:           nginx-deployment-1006230814-6winp
Namespace:      default
Node:           kubernetes-node-wul5/10.240.0.9
Start Time:     Thu, 24 Mar 2016 01:39:49 +0000
Labels:         app=nginx,pod-template-hash=1006230814
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","apiVersion
Status:         Running
IP:             10.244.0.6
Controllers:    ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID:       docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a4
    Image:              nginx
    Image ID:           docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc35
    Port:               80/TCP
    QoS Tier:
      cpu:      Guaranteed
      memory:   Guaranteed
    Limits:
      cpu:      500m
      memory:   128Mi
    Requests:
      memory:           128Mi
      cpu:              500m
    State:              Running
      Started:          Thu, 24 Mar 2016 01:39:51 +0000
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5kdvl (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-4bcbi:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
    Optional:   false
QoS Class:      Guaranteed
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen     LastSeen        Count   From                            Su
  ---------     --------        -----   ----                            --
  54s           54s             1       {default-scheduler }
  54s           54s             1       {kubelet kubernetes-node-wul5}  spec.conta
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.conta
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.conta
  53s           53s             1       {kubelet kubernetes-node-wul5}  spec.conta
```

Here you can see configuration information about the container(s) and Pod (labels, resource requirements, etc.), as well as status information about the container(s) and Pod (state, readiness, restart count, events, etc.).

The container state is one of Waiting, Running, or Terminated. Depending on the state, additional information will be provided – here you can see that for a container in Running state, the system tells you when the container started.

Ready tells you whether the container passed its last readiness probe. (In this case, the container does not have a readiness probe configured; the container is assumed to be ready if no readiness probe is configured.)

Restart Count tells you how many times the container has been restarted; this information can be useful for detecting crash loops in containers that are configured with a restart policy of 'always.'

Currently the only Condition associated with a Pod is the binary Ready condition, which indicates that the pod is able to service requests and should be added to the load balancing pools of all matching services.

Lastly, you see a log of recent events related to your Pod. The system compresses multiple identical events by indicating the first and last time it was seen and the number of times it was seen. "From" indicates the component that is logging the event, "SubobjectPath" tells you which object (e.g. container within the pod) is being referred to, and "Reason" and "Message" tell you what happened.

Example: debugging Pending Pods

A common scenario that you can detect using events is when you've created a Pod that won't fit on any node. For example, the Pod might request more resources than are free on any node, or it might specify a label selector that doesn't match any nodes. Let's say we created the previous Deployment with 5 replicas (instead of 2) and requesting 600 millicores instead of 500, on a four-node cluster where each (virtual) machine has 1 CPU. In that case one of the Pods will not be able to schedule. (Note that because of the cluster addon pods such as fluentd, skydns, etc., that run on each node, if we requested 1000 millicores then none of the Pods would be able to schedule.)

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1       Running   0          7m
nginx-deployment-1006230814-fmgu3   1/1       Running   0          7m
nginx-deployment-1370807587-6ekbw   1/1       Running   0          1m
nginx-deployment-1370807587-fg172   0/1       Pending   0          1m
nginx-deployment-1370807587-fz9sd   0/1       Pending   0          1m
```

To find out why the nginx-deployment-1370807587-fz9sd pod is not running, we can use `kubectl describe pod` on the pending Pod and look at its events:

```
$ kubectl describe pod nginx-deployment-1370807587-fz9sd
  Name:         nginx-deployment-1370807587-fz9sd
  Namespace:    default
  Node:         /
  Labels:               app=nginx,pod-template-hash=1370807587
  Status:               Pending
  IP:
  Controllers:  ReplicaSet/nginx-deployment-1370807587
  Containers:
    nginx:
      Image:    nginx
      Port:     80/TCP
      QoS Tier:
        memory: Guaranteed
        cpu:    Guaranteed
      Limits:
        cpu:    1
        memory: 128Mi
      Requests:
        cpu:    1
        memory: 128Mi
      Environment Variables:
  Volumes:
    default-token-4bcbi:
      Type:     Secret (a volume populated by a Secret)
      SecretName:       default-token-4bcbi
  Events:
    FirstSeen   LastSeen        Count   From                            SubobjectP
    ---------   --------        -----   ----                            ----------
    1m              48s             7       {default-scheduler }
  fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CP
  fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CP
```

Here you can see the event generated by the scheduler saying that the Pod failed to schedule for reason `FailedScheduling` (and possibly others). The message tells us that there were not enough resources for the Pod on any of the nodes.

To correct this situation, you can use `kubectl scale` to update your Deployment to specify four or fewer replicas. (Or you could just leave the one Pod pending, which is harmless.)

Events such as the ones you saw at the end of `kubectl describe pod` are persisted in etcd and provide high-level information on what is happening in the cluster. To list all events you can use

```
kubectl get events
```

but you have to remember that events are namespaced. This means that if you're interested in events for some namespaced object (e.g. what happened with Pods in namespace `my-namespace`) you need to explicitly provide a namespace to the command:

```
kubectl get events --namespace=my-namespace
```

To see events from all namespaces, you can use the `--all-namespaces` argument.

In addition to `kubectl describe pod`, another way to get extra information about a pod (beyond what is provided by `kubectl get pod`) is to pass the `-o yaml` output format flag to `kubectl get pod`. This will give you, in YAML format, even more information than `kubectl describe pod`–essentially all of the information the system has about the Pod. Here you will see things like annotations (which are key-value metadata without the label restrictions, that is used internally by Kubernetes system components), restart policy, ports, and volumes.

```
$kubectl get pod nginx-deployment-1006230814-6winp -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind":"SerializedReference","apiVersion":"v1","reference":{"kind":"Replica
  creationTimestamp: 2016-03-24T01:39:50Z
  generateName: nginx-deployment-1006230814-
  labels:
    app: nginx
    pod-template-hash: "1006230814"
  name: nginx-deployment-1006230814-6winp
  namespace: default
  resourceVersion: "133447"
  selfLink: /api/v1/namespaces/default/pods/nginx-deployment-1006230814-6winp
  uid: 4c879808-f161-11e5-9a78-42010af00005
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    resources:
      limits:
        cpu: 500m
        memory: 128Mi
```

```
          memory: 128Mi
        requests:
          cpu: 500m
          memory: 128Mi
      terminationMessagePath: /dev/termination-log
      volumeMounts:
      - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
        name: default-token-4bcbi
        readOnly: true
    dnsPolicy: ClusterFirst
    nodeName: kubernetes-node-wul5
    restartPolicy: Always
    securityContext: {}
    serviceAccount: default
    serviceAccountName: default
    terminationGracePeriodSeconds: 30
    volumes:
    - name: default-token-4bcbi
      secret:
        secretName: default-token-4bcbi
  status:
    conditions:
    - lastProbeTime: null
      lastTransitionTime: 2016-03-24T01:39:51Z
      status: "True"
      type: Ready
    containerStatuses:
    - containerID: docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae3
      image: nginx
      imageID: docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e516
      lastState: {}
      name: nginx
      ready: true
      restartCount: 0
      state:
        running:
          startedAt: 2016-03-24T01:39:51Z
    hostIP: 10.240.0.9
    phase: Running
    podIP: 10.244.0.6
    startTime: 2016-03-24T01:39:49Z
```

Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node – for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
$ kubectl get nodes
NAME                     STATUS       AGE     VERSION
kubernetes-node-861h     NotReady     1h      v1.6.0+fff5156
kubernetes-node-bols     Ready        1h      v1.6.0+fff5156
kubernetes-node-st6x     Ready        1h      v1.6.0+fff5156
kubernetes-node-unaj     Ready        1h      v1.6.0+fff5156

$ kubectl describe node kubernetes-node-861h
Name:                   kubernetes-node-861h
Role
Labels:          beta.kubernetes.io/arch=amd64
           beta.kubernetes.io/os=linux
           kubernetes.io/hostname=kubernetes-node-861h
Annotations:        node.alpha.kubernetes.io/ttl=0
                    volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:             <none>
CreationTimestamp:      Mon, 04 Sep 2017 17:13:23 +0800
Phase:
Conditions:
  Type          Status          LastHeartbeatTime                      LastTransi
  ----    ------    -----------------     ------------------      ------
  OutOfDisk             Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fr
  MemoryPressure        Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fr
  DiskPressure          Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fr
  Ready                 Unknown         Fri, 08 Sep 2017 16:04:28 +0800         Fr
Addresses:      10.240.115.55,104.197.0.26
Capacity:
 cpu:           2
 hugePages:     0
 memory:        4046788Ki
 pods:          110
Allocatable:
 cpu:           1500m
 hugePages:     0
 memory:        1479263Ki
 pods:          110
System Info:
 Machine ID:                    8e025a21a4254e11b028584d9d8b12c4
 System UUID:                   349075D1-D169-4F25-9F2A-E886850C47E3
 Boot ID:                       5cd18b37-c5bd-4658-94e0-e436d3f110e0
 Kernel Version:                4.4.0-31-generic
 OS Image:                      Debian GNU/Linux 8 (jessie)
 Operating System:              linux
 Architecture:                  amd64
 Container Runtime Version:     docker://1.12.5
 Kubelet Version:               v1.6.9+a3d1dfa6f4335
 Kube-Proxy Version:            v1.6.9+a3d1dfa6f4335
ExternalID:                     15233045891481496305
Non-terminated Pods:            (9 in total)
  Namespace                     Name                                               CP
  ---------                     ----                                               --
......
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits      Memory Requests         Memory Limits
  ------------  ----------      ---------------         -------------
  900m (60%)    2200m (146%)    1009286400 (66%)        5681286400 (375%)
Events:         <none>

$ kubectl get node kubernetes-node-861h -o yaml
apiVersion: v1
kind: Node
metadata:
  creationTimestamp: 2015-07-10T21:32:29Z
  labels:
    kubernetes.io/hostname: kubernetes-node-861h
  name: kubernetes-node-861h
  resourceVersion: "757"
  selfLink: /api/v1/nodes/kubernetes-node-861h
  uid: 2a69374e-274b-11e5-a234-42010af0d969
spec:
  externalID: "15233045891481496305"
  podCIDR: 10.244.0.0/24
  providerID: gce://striped-torus-760/us-central1-b/kubernetes-node-861h
status:
  addresses:
  - address: 10.240.115.55
    type: InternalIP
  - address: 104.197.0.26
    type: ExternalIP
  capacity:
    cpu: "1"
    memory: 3800808Ki
    pods: "100"
  conditions:
  - lastHeartbeatTime: 2015-07-10T21:34:32Z
    lastTransitionTime: 2015-07-10T21:35:15Z
    reason: Kubelet stopped posting node status.
    status: Unknown
    type: Ready
  nodeInfo:
    bootID: 4e316776-b40d-4f78-a4ea-ab0d73390897
    containerRuntimeVersion: docker://Unknown
    kernelVersion: 3.16.0-0.bpo.4-amd64
    kubeProxyVersion: v0.21.1-185-gffc5a86098dc01
    kubeletVersion: v0.21.1-185-gffc5a86098dc01
    machineID: ""
    osImage: Debian GNU/Linux 7 (wheezy)
    systemUUID: ABE5F6B4-D44B-108B-C46A-24CCE16C8B6E
```

What’s next?

Learn about additional debugging tools, including:

- Logging
- Monitoring
- Getting into containers via `exec`
- Connecting to containers via proxies
- Connecting to containers via port forwarding

Auditing

Kubernetes Audit provides a security-relevant chronological set of records documenting the sequence of activities that have affected system by individual users, administrators or other components of the system. It allows cluster administrator to answer the following questions:

- what happened?
- when did it happen?
- who initiated it?
- on what did it happen?
- where was it observed?
- from where was it initiated?
- to where was it going?

Legacy Audit

Kubernetes audit is part of Kube-apiserver logging all requests processed by the server. Each audit log entry contains two lines:

1. The request line containing a unique ID to match the response and request metadata, such as the source IP, requesting user, impersonation information, resource being requested, etc.
2. The response line containing a unique ID matching the request line and the response code.

Example output for `admin` user listing pods in the `default` namespace:

```
2017-03-21T03:57:09.106841886-04:00 AUDIT: id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b5
2017-03-21T03:57:09.108403639-04:00 AUDIT: id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b5
```

Note that Kubernetes 1.8 has switched to use the advanced structured audit log by default. To fallback to this legacy audit, disable the advanced auditing feature using the `AdvancedAuditing` feature gate on the kube-apiserver:

```
--feature-gates=AdvancedAuditing=false
```

Configuration

Kube-apiserver provides the following options which are responsible for configuring where and how audit logs are handled:

- `audit-log-path` - enables the audit log pointing to a file where the requests are being logged to, '-' means standard out.
- `audit-log-maxage` - specifies maximum number of days to retain old audit log files based on the timestamp encoded in their filename.
- `audit-log-maxbackup` - specifies maximum number of old audit log files to retain.
- `audit-log-maxsize` - specifies maximum size in megabytes of the audit log file before it gets rotated. Defaults to 100MB.

If an audit log file already exists, Kubernetes appends new audit logs to that file. Otherwise, Kubernetes creates an audit log file at the location you specified in `audit-log-path`. If the audit log file exceeds the size you specify in `audit-log-maxsize`, Kubernetes will rename the current log file by appending the current timestamp on the file name (before the file extension) and create a new audit log file. Kubernetes may delete old log files when creating a new log file; you can configure how many files are retained and how old they can be by specifying the `audit-log-maxbackup` and `audit-log-maxage` options.

Advanced audit

Kubernetes 1.7 expands auditing with experimental functionality such as event filtering and a webhook for integration with external systems. Kubernetes 1.8 upgrades the advanced audit feature to beta, and some backward incompatible changes have been committed.

`AdvancedAuditing` is customizable in two ways. Policy, which determines what's recorded, and backends, which persist records. Backend implementations include logs files and webhooks.

The structure of audit events changes when enabling the `AdvancedAuditing` feature flag. This includes some cleanups, such as the `method` reflecting the verb evaluated by the authorization layer instead of the HTTP verb. Also, instead of always generating two events per request, events are recorded with an associated "stage". The known stages are:

- `RequestReceived` - The stage for events generated as soon as the audit handler receives the request.
- `ResponseStarted` - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- `ResponseComplete` - Once the response body has been completed.
- `Panic` - Events generated when a panic occurred.

Audit Policy

Audit policy is a document defining rules about what events should be recorded. The policy is passed to the kube-apiserver using the `--audit-policy-file` flag.

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml
```

If `AdvancedAuditing` is enabled and this flag is omitted, no events are logged.

The policy file holds rules that determine the level of an event. Known audit levels are:

- `None` - don't log events that match this rule.
- `Metadata` - log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- `Request` - log event metadata and request body but not response body.
- `RequestResponse` - log event metadata, request and response bodies.

When an event is processed, it's compared against the list of rules in order. The first matching rule sets the audit level of the event. The audit policy is defined by the `audit.k8s.io` API group. Some new fields are supported in beta version, like `resourceNames` and `omitStages`.

In Kubernetes 1.8 `kind` and `apiVersion` along with `rules` **must** be provided in the audit policy file. A policy file with 0 rules, or a policy file that doesn't provide a valid `apiVersion` and `kind` value will be treated as illgal.

Some example audit policy files:

```
apiVersion: audit.k8s.io/v1beta1  #this is required in Kubernetes 1.8
kind: Policy
rules:
  # Don't log watch requests by the "system:kube-proxy" on endpoints or services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs:
    - "/api*" # Wildcard matching.
    - "/version"

  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
    - group: "" # core API group
      resources: ["configmaps"]
    # This rule only applies to resources in the "kube-system" namespace.
    # The empty string "" can be used to select non-namespaced resources.
    namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at the Metadata level
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]

  # Log all other resources in core and extensions at the Request level.
  - level: Request
    resources:
    - group: "" # core API group
    - group: "extensions" # Version of group should NOT be included.

  # A catch-all rule to log all other requests at the Metadata level.
  - level: Metadata
```

The next audit policy file shows new features introduced in Kubernetes 1.8:

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  # Log pod changes at Request level
  - level: Request
    resources:
    - group: ""
      # Resource "pods" no longer matches requests to any subresource of pods,
      # This behavior is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
      resourceNames: ["controller-leader"]

  # A catch-all rule to log all other requests at the Metadata level.
  # For this rule we use "omitStages" to omit events at "ReqeustReceived" stage.
  # Events in this stage will not be sent to backend.
  - level: Metadata
    omitStages:
      - "RequestReceived"
```

You can use a minimal audit policy file to log all requests at the `Metadata` level:

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
- level: Metadata
```

The audit profile used by GCE should be used as reference by admins constructing their own audit profiles.

Audit backends

Audit backends implement strategies for emitting events. The kube-apiserver provides a logging and webhook backend.

Each request to the API server can generate multiple events, one when the request is received, another when the response is sent, and additional events for long running requests (such as watches). The ID of events will be the same if they were generated from the same request.

The event format is defined by the `audit.k8s.io` API group. The `v1alpha1` format of this API can be found here with more details about the exact fields captured.

Log backend

The behavior of the `--audit-log-path` flag changes when enabling the `AdvancedAuditing` feature flag. All generated events defined by `--audit-policy-file` are recorded in structured json format:

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1beta1","metadata":{"creationTimestamp
{"kind":"Event","apiVersion":"audit.k8s.io/v1beta1","metadata":{"creationTimestamp
```

In alpha version, objectRef.apiVersion holds both the api group and version. In beta version these were break out into objectRef.apiGroup and objectRef.apiVersion.

Starting from Kubernetes 1.8, structured json format is used for log backend by default. Use the following option to switch log to legacy format:

```
--audit-log-format=legacy
```

With legacy format, events are formatted as follows:

```
2017-09-05T06:08:19.885328047-04:00 AUDIT: id="c28a95ad-f9dd-47e1-a617-b6dc152db95
2017-09-05T06:08:19.885328047-04:00 AUDIT: id="c28a95ad-f9dd-47e1-a617-b6dc152db95
```

Logged events omit the request and response bodies. The `Request` and `RequestResponse` levels are equivalent to `Metadata` for legacy format. This legacy format of advanced audit is different from the Legacy Audit discussed above, such as changes to the method values and the introduction of a “stage” for each event.

Webhook backend

The audit webhook backend can be used to have kube-apiserver send audit events to a remote service. The webhook requires the `AdvancedAuditing` feature flag and is configured using the following command line flags:

```
--audit-webhook-config-file=/etc/kubernetes/audit-webhook-kubeconfig
--audit-webhook-mode=batch
```

`audit-webhook-mode` controls buffering strategies used by the webhook. Known modes are:

- `batch` - buffer events and asynchronously send the set of events to the external service.
- `blocking` - block API server responses on sending each event to the external service.

The webhook config file uses the kubeconfig format to specify the remote address of the service and credentials used to connect to it.

```
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-audit-service
    cluster:
      certificate-authority: /path/to/ca.pem  # CA for verifying the remote servic
      server: https://audit.example.com/audit # URL of remote service to query. Mu

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook plugin to use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API server.
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-audit-service
    user: name-of-api-sever
  name: webhook
```

Events are POSTed as a JSON serialized `EventList`. An example payload:

```
{
    "apiVersion": "audit.k8s.io/v1beta1",
    "items": [
        {
            "auditID": "24f30caf-d7d4-45d5-b7bd-e7af300d7886",
            "level": "Metadata",
            "metadata": {
                "creationTimestamp": null
            },
            "objectRef": {
                "apiGroup": "rbac.authorization.k8s.io",
                "apiVersion": "v1",
                "name": "jane",
                "namespace": "default",
                "resource": "roles"
            },
            "requestURI": "/apis/rbac.authorization.k8s.io/v1/namespaces/default/r
            "responseStatus": {
                "code": 200,
                "metadata": {}
            },
            "sourceIPs": [
                "172.16.116.128"
            ],
            "stage": "ResponseComplete",
            "timestamp": "2017-09-05T10:20:24Z",
            "user": {
                "groups": [
                    "system:masters",
                    "system:authenticated"
                ],
                "username": "kubecfg"
            },
            "verb": "get"
        }
    ],
    "kind": "EventList",
    "metadata": {}
}
```

Audit-Id

Audit-Id is a unique ID for each http request to kube-apiserver. The ID of events will be the same if they were generated from the same request. Starting from Kubernetes 1.8, if an audit event is generated for the request, kube-apiserver will respond with an Audit-Id in the HTTP header. Note that for some special requests like `kubectl exec`, `kubectl attach`, kube-apiserver works like a proxy, no Audit-Id will be returned even if audit events are recorded.

Log Collector Examples

Use fluentd to collect and distribute audit events from log file

Fluentd is an open source data collector for unified logging layer. In this example, we will use fluentd to split audit events by different namespaces. Note that this example requries json format output support in Kubernetes 1.8.

1. install fluentd, fluent-plugin-forest and fluent-plugin-rewrite-tag-filter in the kube-apiserver node
2. create a config file for fluentd

```
$ cat <<EOF > /etc/fluentd/config
# fluentd conf runs in the same host with kube-apiserver
<source>
    @type tail
    # audit log path of kube-apiserver
    path /var/log/audit
    pos_file /var/log/audit.pos
    format json
    time_key time
    time_format %Y-%m-%dT%H:%M:%S.%N%z
    tag audit
</source>

<filter audit>
    #https://github.com/fluent/fluent-plugin-rewrite-tag-filter/issues/13
    type record_transformer
    enable_ruby
    <record>
     namespace ${record["objectRef"].nil? ? "none":(record["objectRef"]["na
    </record>
</filter>
```

```
<match audit>
    # route audit according to namespace element in context
    @type rewrite_tag_filter
    rewriterule1 namespace ^(.+) ${tag}.$1
</match>

<filter audit.**>
       @type record_transformer
       remove_keys namespace
</filter>

<match audit.**>
    @type forest
    subtype file
    remove_prefix audit
    <template>
        time_slice_format %Y%m%d%H
        compress gz
        path /var/log/audit-${tag}.*.log
        format json
        include_time_key true
    </template>
</match>
```

3. start fluentd

```
$ fluentd -c /etc/fluentd/config  -vv
```

4. start kube-apiserver with the following options:

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-log-path=/var
```

5. check audits for different namespaces in /var/log/audit-*.log

Use logstash to collect and distribute audit events from webhook backend

Logstash is an open source, server-side data processing tool. In this example, we will use logstash to collect audit events from webhook backend, and save events of different users into different files.

1. install logstash
2. create config file for logstash

```
$ cat <<EOF > /etc/logstash/config
input{
    http{
        #TODO, figure out a way to use kubeconfig file to authenticate to lo
        #https://www.elastic.co/guide/en/logstash/current/plugins-inputs-ht
        port=>8888
    }
}
filter{
    split{
        # Webhook audit backend sends several events together with EventLis
        # split each event here.
        field=>[items]
        # We only need event subelement, remove others.
        remove_field=>[headers, metadata, apiVersion, "@timestamp", kind, "@
    }
    mutate{
        rename => {items=>event}
    }
}
output{
    file{
        # Audit events from different users will be saved into different fi
        path=>"/var/log/kube-audit-%{[event][user][username]}/audit"
    }
}
```

3. start logstash

```
$ bin/logstash -f /etc/logstash/config --path.settings /etc/logstash/
```

4. create a kubeconfig file for kube-apiserver webhook audit backend

```
$ cat <<EOF > /etc/kubernetes/audit-webhook-kubeconfig
apiVersion: v1
clusters:
- cluster:
    server: http://<ip_of_logstash>:8888
  name: logstash
contexts:
- context:
    cluster: logstash
    user: ""
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users: []
EOF
```

5. start kube-apiserver with the following options:

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-webhook-confi
```

6. check audits in logstash node's directories /var/log/kube-audit-*/audit

Note that in addition to file output plugin, logstash has a variety of outputs that let users route data where they want. For example, users can emit audit events to elasticsearch plugin which supports full-text search and analytics.

Use an HTTP Proxy to Access the Kubernetes API

This page shows how to use an HTTP proxy to access the Kubernetes API.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- If you do not already have an application running in your cluster, start a Hello world application by entering this command:

```
kubectl run node-hello --image=gcr.io/google-samples/node-hello:1.0 --port=808
```

Using kubectl to start a proxy server

This command starts a proxy to the Kubernetes API server:

```
kubectl proxy --port=8080
```

Exploring the Kubernetes API

When the proxy server is running, you can explore the API using `curl`, `wget`, or a browser.

Get the API versions:

```
curl http://localhost:8080/api/

{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.2.15:8443"
    }
  ]
}
```

Get a list of pods:

```
curl http://localhost:8080/api/v1/namespaces/default/pods

{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "33074"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes-bootcamp-2321272333-ix8pt",
        "generateName": "kubernetes-bootcamp-2321272333-",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/kubernetes-bootcamp-232127233
        "uid": "ba21457c-6b1d-11e6-85f7-1ef9f1dab92b",
        "resourceVersion": "33003",
        "creationTimestamp": "2016-08-25T23:43:30Z",
        "labels": {
          "pod-template-hash": "2321272333",
          "run": "kubernetes-bootcamp"
        },
        ...
}
```

What’s next

Learn more about kubectl proxy.

Extend the Kubernetes API with CustomResourceDefinitions

This page shows how to install a custom resource into the Kubernetes API by creating a CustomResourceDefinition.

Before you begin

- Read about custom resources.
- Make sure your Kubernetes cluster has a master version of 1.7.0 or higher.

Create a CustomResourceDefinition

When you create a new *CustomResourceDefinition* (CRD), the Kubernetes API Server reacts by creating a new RESTful resource path, either namespaced or cluster-scoped, as specified in the CRD's `scope` field. As with existing built-in objects, deleting a namespace deletes all custom objects in that namespace. CustomResourceDefinitions themselves are non-namespaced and are available to all namespaces.

For example, if you save the following CustomResourceDefinition to `resourcedefinition.yaml`:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # version name to use for REST API: /apis/<group>/<version>
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource manifests use t
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - ct
```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

Then a new namespaced RESTful API endpoint is created at:

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

This endpoint URL can then be used to create and manage custom objects. The `kind` of these objects will be `CronTab` from the spec of the CustomResourceDefinition object you created above.

Create custom objects

After the CustomResourceDefinition object has been created, you can create custom objects. Custom objects can contain custom fields. These fields can contain arbitrary JSON. In the following example, the `cronSpec` and `image` custom fields are set in a custom object of kind `CronTab`. The kind `CronTab` comes from the spec of the CustomResourceDefinition object you created above.

If you save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

and create it:

```
kubectl create -f my-crontab.yaml
```

You can then manage your CronTab objects using kubectl. For example:

```
kubectl get crontab
```

Should print a list like this:

```
NAME                   KIND
my-new-cron-object     CronTab.v1.stable.example.com
```

Note that resource names are not case-sensitive when using kubectl, and you can use either the singular or plural forms defined in the CRD, as well as any short names.

You can also view the raw YAML data:

```
kubectl get ct -o yaml
```

You should see that it contains the custom `cronSpec` and `image` fields from the yaml you used to create it:

```
apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * */5'
    image: my-awesome-cron-image
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

Advanced topics

Finalizers

Finalizers allow controllers to implement asynchronous pre-delete hooks. Custom objects support finalizers just like built-in objects.

You can add a finalizer to a custom object like this:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
  - finalizer.stable.example.com
```

The first delete request on an object with finalizers merely sets a value for the `metadata.deletionTimestamp` field instead of deleting it. This triggers controllers watching the object to execute any finalizers they handle.

Each controller then removes its finalizer from the list and issues the delete request again. This request only deletes the object if the list of finalizers is now empty, meaning all finalizers are done.

Validation

Validation of custom objects is possible via OpenAPI v3 schema. Additionally, the following restrictions are applied to the schema:

- The fields `default`, `nullable`, `discriminator`, `readOnly`, `writeOnly`, `xml` and `deprecated` cannot be set.
- The field `uniqueItems` cannot be set to true.
- The field `additionalProperties` cannot be set to false.

This feature is **alpha** in v1.8 and may change in backward incompatible ways. Enable this feature using the `CustomResourceValidation` feature gate on the kube-apiserver:

```
--feature-gates=CustomResourceValidation=true
```

The schema is defined in the CustomResourceDefinition. In the following example, the CustomResourceDefinition applies the following validations on the custom object:

- `spec.cronSpec` must be a string and must be of the form described by the regular expression.
- `spec.replicas` must be an integer and must have a minimum value of 1 and a maximum value of 10.

Save the CustomResourceDefinition to `resourcedefinition.yaml`:

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
  validation:
   # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      properties:
        spec:
          properties:
            cronSpec:
              type: string
              pattern: '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+)?){4}$'
            replicas:
              type: integer
              minimum: 1
              maximum: 10
```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

A request to create a custom object of kind `CronTab` will be rejected if there are invalid values in its fields. In the following example, the custom object contains fields with invalid values:

- `spec.cronSpec` does not match the regular expression.
- `spec.replicas` is greater than 10.

If you save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * *"
  image: my-awesome-cron-image
  replicas: 15
```

and create it:

```
kubectl create -f my-crontab.yaml
```

you will get an error:

```
The CronTab "my-new-cron-object" is invalid: []: Invalid value: map[string]interfa
validation failure list:
spec.cronSpec in body should match '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+)?){4}$'
spec.replicas in body should be less than or equal to 10
```

If the fields contain valid values, the object creation request is accepted.

Save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
  replicas: 5
```

And create it:

```
kubectl create -f my-crontab.yaml
crontab "my-new-cron-object" created
```

What's next

- Learn how to Migrate a ThirdPartyResource to CustomResourceDefinition.

Extend the Kubernetes API with ThirdPartyResources

DEPRECATION NOTICE: As of `Kubernetes 1.7`, this has been deprecated

What is ThirdPartyResource?

ThirdPartyResource is deprecated as of Kubernetes 1.7 and has been removed in version 1.8 in accordance with the deprecation policy for beta features.

To avoid losing data stored in ThirdPartyResources, you must migrate to CustomResourceDefinition before upgrading to Kubernetes 1.8 or higher.

Kubernetes comes with many built-in API objects. However, there are often times when you might need to extend Kubernetes with your own API objects in order to do custom automation.

`ThirdPartyResource` objects are a way to extend the Kubernetes API with a new API object type. The new API object type will be given an API endpoint URL and support CRUD operations, and watch API. You can then create custom objects using this API endpoint. You can think of `ThirdPartyResources` as being much like the schema for a database table. Once you have created the table, you can then start storing rows in the table. Once created, `ThirdPartyResources` can act as the data model behind custom controllers or automation programs.

Structure of a ThirdPartyResource

Each `ThirdPartyResource` has the following:

- `metadata` - Standard Kubernetes object metadata.
- `kind` - The kind of the resources described by this third party resource.
- `description` - A free text description of the resource.
- `versions` - A list of the versions of the resource.

The `kind` for a `ThirdPartyResource` takes the form `<kind name>.<domain>`. You are expected to provide a unique kind and domain name in order to avoid conflicts with other `ThirdPartyResource` objects. Kind names will be converted to CamelCase when creating instances of the `ThirdPartyResource`. Hyphens in the `kind` are assumed to be word breaks. For instance the kind `camel-case` would be converted to `CamelCase` but `camelcase` would be converted to `Camelcase`.

Other fields on the `ThirdPartyResource` are treated as custom data fields. These fields can hold arbitrary JSON data and have any structure.

You can view the full documentation about `ThirdPartyResources` using the `explain` command in kubectl.

```
$ kubectl explain thirdpartyresource
```

Creating a ThirdPartyResource

When you create a new `ThirdPartyResource`, the Kubernetes API Server reacts by creating a new, namespaced RESTful resource path. For now, non-namespaced objects are not supported. As with existing built-in objects, deleting a namespace deletes all custom objects in that namespace. `ThirdPartyResources` themselves are non-namespaced and are available to all namespaces.

For example, if you save the following `ThirdPartyResource` to `resource.yaml`:

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: cron-tab.stable.example.com
description: "A specification of a Pod to run on a cron style schedule"
versions:
- name: v1
```

And create it:

```
$ kubectl create -f resource.yaml
thirdpartyresource "cron-tab.stable.example.com" created
```

Then a new RESTful API endpoint is created at:

`/apis/stable.example.com/v1/namespaces/<namespace>/crontabs/...`

This endpoint URL can then be used to create and manage custom objects. The `kind` of these objects will be `CronTab` following the camel case rules applied to the `metadata.name` of this `ThirdPartyResource` (`cron-tab.stable.example.com`)

Creating Custom Objects

After the `ThirdPartyResource` object has been created you can create custom objects. Custom objects can contain custom fields. These fields can contain arbitrary JSON. In the following example, a `cronSpec` and `image` custom fields are set to the custom object of kind `CronTab`. The kind `CronTab` is derived from the `metadata.name` of the `ThirdPartyResource` object we created above.

If you save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
cronSpec: "* * * * /5"
image: my-awesome-cron-image
```

and create it:

```
$ kubectl create -f my-crontab.yaml
crontab "my-new-cron-object" created
```

You can then manage our `CronTab` objects using kubectl. Note that resource names are not case-sensitive when using kubectl:

```
$ kubectl get crontab
NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

You can also view the raw JSON data. Here you can see that it contains the custom `cronSpec` and `image` fields from the yaml you used to create it:

```
$ kubectl get crontab -o json
{
    "apiVersion": "v1",
    "items": [
        {
            "apiVersion": "stable.example.com/v1",
            "cronSpec": "* * * * /5",
            "image": "my-awesome-cron-image",
            "kind": "CronTab",
            "metadata": {
                "creationTimestamp": "2016-09-29T04:59:00Z",
                "name": "my-new-cron-object",
                "namespace": "default",
                "resourceVersion": "12601503",
                "selfLink": "/apis/stable.example.com/v1/namespaces/default/cronta
                "uid": "6f65e7a3-8601-11e6-a23e-42010af0000c"
            }
        }
    ],
    "kind": "List",
    "metadata": {},
    "resourceVersion": "",
    "selfLink": ""
}
```

Migrate a ThirdPartyResource to CustomResourceDefinition

This page shows how to migrate data stored in a ThirdPartyResource (TPR) to a CustomResourceDefinition (CRD).

Kubernetes does not automatically migrate existing TPRs. This is due to API changes introduced as part of graduating to beta under a new name and API group. Instead, both TPR and CRD are available and operate independently in Kubernetes 1.7. Users must migrate each TPR one by one to preserve their data before upgrading to Kubernetes 1.8.

The simplest way to migrate is to stop all clients that use a given TPR, then delete the TPR and start from scratch with a CRD. This page describes an optional process that eases the transition by migrating existing TPR data for you **on a best-effort basis**.

Before you begin

- Make sure your Kubernetes cluster has a **master version of exactly 1.7.x** (any patch release), as this is the only version that supports both TPR and CRD.
- If you use a TPR-based custom controller, check with the author of the controller first. Some or all of these steps may be unnecessary if the custom controller handles the migration for you.
- Be familiar with the concept of custom resources, which were known as *third-party resources* until Kubernetes 1.7.
- Be familiar with CustomResourceDefinitions, which are a simple way to implement custom resources.
- **Before performing a migration on real data, conduct a dry run by going through these steps in a test cluster.**

Migrate TPR data

1. **Rewrite the TPR definition**

   Clients that access the REST API for your custom resource should not need any changes. However, you will need to rewrite your TPR definition as a CRD.

   Make sure you specify values for the CRD fields that match what the server used to fill in for you with TPR.

   For example, if your ThirdPartyResource looks like this:

   ```
   apiVersion: extensions/v1beta1
   kind: ThirdPartyResource
   metadata:
     name: cron-tab.stable.example.com
   description: "A specification of a Pod to run on a cron style schedule"
   versions:
   - name: v1
   ```

   A matching CustomResourceDefinition could look like this:

   ```
   apiVersion: apiextensions.k8s.io/v1beta1
   kind: CustomResourceDefinition
   metadata:
     name: crontabs.stable.example.com
   spec:
     scope: Namespaced
     group: stable.example.com
     version: v1
     names:
       kind: CronTab
       plural: crontabs
       singular: crontab
   ```

2. **Install the CustomResourceDefinition**

   While the source TPR is still active, install the matching CRD with `kubectl create`. Existing TPR data remains accessible because TPRs take precedence over CRDs when both try to serve the same resource.

   After you create the CRD, make sure the *Established* condition goes to True. You can check it with a command like this:

   ```
   kubectl get crd -o 'custom-columns=NAME:{.metadata.name},ESTABLISHED:{.status.
   ```

   The output should look like this:

   ```
   NAME                          ESTABLISHED
   crontabs.stable.example.com   True
   ```

3. **Stop all clients that use the TPR**

   The API server attempts to prevent TPR data for the resource from changing while it copies objects to the CRD, but it can't guarantee consistency in all cases, such as with multiple masters. Stopping clients, such as TPR-based custom controllers, helps to avoid inconsistencies in the copied data.

   In addition, clients that watch TPR data do not receive any more events once the migration begins. You must restart them after the migration completes so they start watching CRD data instead.

4. **Back up TPR data**

   In case the data migration fails, save a copy of existing data for the resource:

   ```
   kubectl get crontabs --all-namespaces -o yaml > crontabs.yaml
   ```

   You should also save a copy of the TPR definition if you don't have one already:

   ```
   kubectl get thirdpartyresource cron-tab.stable.example.com -o yaml --export >
   ```

5. **Delete the TPR definition**

   Normally, when you delete a TPR definition, the API server tries to clean up any objects stored in that resource. Because a matching CRD exists, the server copies objects to the CRD instead of deleting them.

   ```
   kubectl delete thirdpartyresource cron-tab.stable.example.com
   ```

6. **Verify the new CRD data**

   It can take up to 10 seconds for the TPR controller to notice when you delete the TPR definition and to initiate the migration. The TPR data remains accessible during this time.

   Once the migration completes, the resource begins serving through the CRD. Check that all your objects were correctly copied:

   ```
   kubectl get crontabs --all-namespaces -o yaml
   ```

   If the copy failed, you can quickly revert to the set of objects that existed just before the migration by recreating the TPR definition:

   ```
   kubectl create -f tpr.yaml
   ```

7. **Restart clients**

   After verifying the CRD data, restart any clients you stopped before the migration, such as custom controllers and other watchers. These clients now access CRD data when they make requests on the same API endpoints that the TPR previously served.

What’s next

- Learn more about custom resources.
- Learn more about using CustomResourceDefinitions.

Configure the aggregation layer

Configuring the aggregation layer allows the Kubernetes apiserver to be extended with additional APIs, which are not part of the core Kubernetes APIs.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Note: There are a few setup requirements for getting the aggregation layer working in your environment to support mutual TLS auth between the proxy and extension apiservers. Kubernetes and the kube-apiserver have multiple CAs, so make sure that the proxy is signed by the aggregation layer CA and not by something else, like the master CA.

Enable apiserver flags

Enable the aggregation layer via the following kube-apiserver flags. They may have already been taken care of by your provider.

```
--requestheader-client-ca-file=<path to aggregator CA cert>
--requestheader-allowed-names=aggregator
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
--proxy-client-cert-file=<path to aggregator proxy cert>
--proxy-client-key-file=<path to aggregator proxy key>
```

If you are not running kube-proxy on a host running the API server, then you must make sure that the system is enabled with the following apiserver flag:

```
--enable-aggregator-routing=true
```

What’s next

- Setup an extension api-server to work with the aggregation layer.
- For a high level overview, see Extending the Kubernetes API with the aggregation layer.
- Learn how to Extend the Kubernetes API Using Custom Resource Definitions.

Setup an extension API server

Setting up an extension API server to work the aggregation layer allows the Kubernetes apiserver to be extended with additional APIs, which are not part of the core Kubernetes APIs.

Before you begin

- You need to have a Kubernetes cluster running.
- You must configure the aggregation layer and enable the apiserver flags.

Setup an extension api-server to work with the aggregation layer

The following steps describe how to set up an extension-apiserver *at a high level*. For a concrete example of how they can be implemented, you can look at the sample-apiserver in the Kubernetes repo.

Alternatively, you can use an existing 3rd party solution, such as apiserver-builder, which should generate a skeleton and automate all of the following steps for you.

1. Make sure the APIService API is enabled (check `--runtime-config`). It should be on by default, unless it's been deliberately turned off in your cluster.
2. You may need to make an RBAC rule allowing you to add APIService objects, or get your cluster administrator to make one. (Since API extensions affect the entire cluster, it is not recommended to do testing/development/debug of an API extension in a live cluster.)
3. Create the Kubernetes namespace you want to run your extension api-service in.

4. Create/get a CA cert to be used to sign the server cert the extension api-server uses for HTTPS.
5. Create a server cert/key for the api-server to use for HTTPS. This cert should be signed by the above CA. It should also have a CN of the Kube DNS name. This is derived from the Kubernetes service and be of the form ..svc
6. Create a Kubernetes secret with the server cert/key in your namespace.
7. Create a Kubernetes deployment for the extension api-server and make sure you are loading the secret as a volume. It should contain a reference to a working image of your extension api-server. The deployment should also be in your namespace.
8. Make sure that your extension-apiserver loads those certs from that volume and that they are used in the HTTPS handshake.
9. Create a Kubernetes service account in your namespace.
10. Create a Kubernetes cluster role for the operations you want to allow on your resources.
11. Create a Kubernetes cluster role binding from the default service account in your namespace to the cluster role you just created.
12. Create a Kubernetes apiservice. The CA cert above should be base 64 encoded, stripped of new lines and used as the spec.caBundle in the apiservce. This should not be namespaced.
13. Use kubectl to get your resource. It should return “No resources found.” Which means that everything worked but you currently have no objects of that resource type created yet.

What’s next

- If you haven’t already, configure the aggregation layer and enable the apiserver flags.
- For a high level overview, see Extending the Kubernetes API with the aggregation layer.
- Learn how to Extend the Kubernetes API Using Custom Resource Definitions.

Manage TLS Certificates in a Cluster

Overview

Every Kubernetes cluster has a cluster root Certificate Authority (CA). The CA is generally used by cluster components to validate the API server's certificate, by the API server to validate kubelet client certificates, etc. To support this, the CA certificate bundle is distributed to every node in the cluster and is distributed as a secret attached to default service accounts. Optionally, your workloads can use this CA to establish trust. Your application can request a certificate signing using the `certificates.k8s.io` API using a protocol that is similar to the ACME draft.

Trusting TLS in a Cluster

Trusting the cluster root CA from an application running as a pod usually requires some extra application configuration. You will need to add the CA certificate bundle to the list of CA certificates that the TLS client or server trusts. For example, you would do this with a golang TLS config by parsing the certificate chain and adding the parsed certificates to the `Certificates` field in the `tls.Config` struct.

The CA certificate bundle is automatically mounted into pods using the default service account at the path `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`. If you are not using the default service account, ask a cluster administrator to build a configmap containing the certificate bundle that you have access to use.

Requesting a Certificate

The following section demonstrates how to create a TLS certificate for a Kubernetes service accessed through DNS.

Step 0. Download and install CFSSL

The cfssl tools used in this example can be downloaded at https://pkg.cfssl.org/.

Step 1. Create a Certificate Signing Request

Generate a private key and certificate signing request (or CSR) by running the following command:

```
$ cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "172.168.0.24",
    "10.0.34.2"
  ],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

Where `172.168.0.24` is the service's cluster IP, `my-svc.my-namespace.svc.cluster.local` is the service's DNS name, `10.0.34.2` is the pod's IP and `my-pod.my-namespace.pod.cluster.local` is the pod's DNS name. You should see the following output:

```
2017/03/21 06:48:17 [INFO] generate received request
2017/03/21 06:48:17 [INFO] received CSR
2017/03/21 06:48:17 [INFO] generating key: ecdsa-256
2017/03/21 06:48:17 [INFO] encoded CSR
```

This command generates two files; it generates `server.csr` containing the PEM encoded pkcs#10 certification request, and `server-key.pem` containing the PEM encoded key to the certificate that is still to be created.

Step 2. Create a Certificate Signing Request object to send to the Kubernetes API

Generate a CSR yaml blob and send it to the apiserver by running the following command:

```
$ cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  groups:
  - system:authenticated
  request: $(cat server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

Notice that the `server.csr` file created in step 1 is base64 encoded and stashed in the `.spec.request` field. We are also requesting a certificate with the “digital signature”, “key encipherment”, and “server auth” key usages. We support all key usages and extended key usages listed here so you can request client certificates and other certificates using this same API.

The CSR should now be visible from the API in a Pending state. You can see it by running:

```
$ kubectl describe csr my-svc.my-namespace
Name:                   my-svc.my-namespace
Labels:                 <none>
Annotations:            <none>
CreationTimestamp:      Tue, 21 Mar 2017 07:03:51 -0700
Requesting User:        yourname@example.com
Status:                 Pending
Subject:
        Common Name:    my-svc.my-namespace.svc.cluster.local
        Serial Number:
Subject Alternative Names:
        DNS Names:      my-svc.my-namespace.svc.cluster.local
        IP Addresses:   172.168.0.24
                        10.0.34.2
Events: <none>
```

Step 3. Get the Certificate Signing Request Approved

Approving the certificate signing request is either done by an automated approval process or on a one off basis by a cluster administrator. More information on what this involves is covered below.

Step 4. Download the Certificate and Use It

Once the CSR is signed and approved you should see the following:

```
$ kubectl get csr
NAME                  AGE       REQUESTOR               CONDITION
my-svc.my-namespace   10m       yourname@example.com    Approved,Issued
```

You can download the issued certificate and save it to a `server.crt` file by running the following:

```
$ kubectl get csr my-svc.my-namespace -o jsonpath='{.status.certificate}' \
    | base64 -d > server.crt
```

Now you can use `server.crt` and `server-key.pem` as the keypair to start your HTTPS server.

Approving Certificate Signing Requests

A Kubernetes administrator (with appropriate permissions) can manually approve (or deny) Certificate Signing Requests by using the `kubectl certificate approve` and `kubectl certificate deny` commands. However if you intend to make heavy usage of this API, you might consider writing an automated certificates controller.

Whether a machine or a human using kubectl as above, the role of the approver is to verify that the CSR satisfies two requirements:

1. The subject of the CSR controls the private key used to sign the CSR. This addresses the threat of a third party masquerading as an authorized subject. In the above example, this step would be to verify that the pod controls the private key used to generate the CSR.
2. The subject of the CSR is authorized to act in the requested context. This addresses the threat of an undesired subject joining the cluster. In the above example, this step would be to verify that the pod is allowed to participate in the requested service.

If and only if these two requirements are met, the approver should approve the CSR and otherwise should deny the CSR.

A Word of Warning on the Approval Permission

The ability to approve CSRs decides who trusts who within the cluster. This includes who the Kubernetes API trusts. The ability to approve CSRs should not be granted broadly or lightly. The requirements of the challenge noted in the previous section and the repercussions of issuing a specific certificate should be fully understood before granting this permission. See here for information on how certificates interact with authentication.

A Note to Cluster Administrators

This tutorial assumes that a signer is setup to serve the certificates API. The Kubernetes controller manager provides a default implementation of a signer. To enable it, pass the `--cluster-signing-cert-file` and `--cluster-signing-key-file` parameters to the controller manager with paths to your Certificate Authority's keypair.

Certificate Rotation

This page shows how to enable and configure certificate rotation for the kubelet.

Before you begin

- Kubernetes version 1.8.0 or later is required
- Kubelet certificate rotation is beta in 1.8.0 which means it may change without notice.

Overview

The kubelet uses certificates for authenticating to the Kubernetes API. By default, these certificates are issued with one year expiration so that they do not need to be renewed too frequently.

Kubernetes 1.8 contains kubelet certificate rotation, a beta feature that will automatically generate a new key and request a new certificate from the Kubernetes API as the current certificate approaches expiration. Once the new certificate is available, it will be used for authenticating connections to the Kubernetes API.

Enabling client certificate rotation

The `kubelet` process accepts an argument `--rotate-certificates` that controls if the kubelet will automatically request a new certificate as the expiration of the certificate currently in use approaches. Since certificate rotation is a beta feature, the feature flag must also be enabled with `--feature-gates=RotateKubeletClientCertificate=true` .

The `kube-controller-manager` process accepts an argument `--experimental-cluster-signing-duration` that controls how long certificates will be issued for.

Understanding the certificate rotation configuration

When a kubelet starts up, if it is configured to bootstrap (using the `--bootstrap-kubeconfig` flag), it will use its initial certificate to connect to the Kubernetes API and issue a certificate signing request. You can view the status of certificate signing requests using:

```
kubectl get csr
```

Initially a certificate signing request from the kubelet on a node will have a status of `Pending`. If the certificate signing requests meets specific criteria, it will be auto approved by the controller manager, then it will have a status of `Approved`. Next, the controller manager will sign a certificate, issued for the duration specified by the `--experimental-cluster-signing-duration` parameter, and the signed certificate will be attached to the certificate signing requests.

The kubelet will retrieve the signed certificate from the Kubernetes API and write that to disk, in the location specified by `--cert-dir`. Then the kubelet will use the new certificate to connect to the Kubernetes API.

As the expiration of the signed certificate approaches, the kubelet will automatically issue a new certificate signing request, using the Kubernetes API. Again, the controller manager will automatically approve the certificate request and attach a signed certificate to the certificate signing request. The kubelet will retrieve the new signed certificate from the Kubernetes API and write that to disk. Then it will update the connections it has to the Kubernetes API to reconnect using the new certificate.

Configure Default Memory Requests and Limits for a Namespace

This page shows how to configure default memory requests and limits for a namespace. If a Container is created in a namespace that has a default memory limit, and the Container does not specify its own memory limit, then the Container is assigned the default memory limit. Kubernetes assigns a default memory request under certain conditions that are explained later in this topic.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 300 GiB of memory.

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace default-mem-example
```

Create a LimitRange and a Pod

Here’s the configuration file for a LimitRange object. The configuration specifies a default memory request and a default memory limit.

memory-defaults.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
    type: Container
```

Create the LimitRange in the default-mem-example namespace:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults.yam
```

Now if a Container is created in the default-mem-example namespace, and the Container does not specify its own values for memory request and memory limit, the Container is given a default memory request of 256 MiB and a default memory limit of 512 MiB.

Here’s the configuration file for a Pod that has one Container. The Container does not specify a memory request and limit.

memory-defaults-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo
spec:
  containers:
  - name: default-mem-demo-ctr
    image: nginx
```

Create the Pod.

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod
```

View detailed information about the Pod:

```
kubectl get pod default-mem-demo --output=yaml --namespace=default-mem-example
```

The output shows that the Pod's Container has a memory request of 256 MiB and a memory limit of 512 MiB. These are the default values specified by the LimitRange.

```yaml
containers:
- image: nginx
  imagePullPolicy: Always
  name: default-mem-demo-ctr
  resources:
    limits:
      memory: 512Mi
    requests:
      memory: 256Mi
```

Delete your Pod:

```
kubectl delete pod default-mem-demo --namespace=default-mem-example
```

What if you specify a Container's limit, but not its request?

Here's the configuration file for a Pod that has one Container. The Container specifies a memory limit, but not a request:

memory-defaults-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-2
spec:
  containers:
  - name: defalt-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1Gi"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod
```

View detailed information about the Pod:

```
kubectl get pod default-mem-demo-2 --output=yaml --namespace=default-mem-example
```

The output shows that the Container's memory request is set to match its memory limit. Notice that the Container was not assigned the default memory request value of 256Mi.

```
resources:
  limits:
    memory: 1Gi
  requests:
    memory: 1Gi
```

What if you specify a Container’s request, but not its limit?

Here’s the configuration file for a Pod that has one Container. The Container specifies a memory request, but not a limit:

memory-defaults-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-3
spec:
  containers:
  - name: default-mem-demo-3-ctr
    image: nginx
    resources:
      requests:
        memory: "128Mi"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod
```

View the Pod’s specification:

```
kubectl get pod default-mem-demo-3 --output=yaml --namespace=default-mem-example
```

The output shows that the Container’s memory request is set to the value specified in the Container’s configuration file. The Container’s memory limit is set to 512Mi, which is the default memory limit for the namespace.

```
resources:
  limits:
    memory: 512Mi
  requests:
    memory: 128Mi
```

Motivation for default memory limits and requests

If your namespace has a resource quota, it is helpful to have a default value in place for memory limit. Here are two of the restrictions that a resource quota imposes on a namespace:

- Every Container that runs in the namespace must have its own memory limit.
- The total amount of memory used by all Containers in the namespace must not exceed a specified limit.

If a Container does not specify its own memory limit, it is given the default limit, and then it can be allowed to run in a namespace that is restricted by a quota.

What’s next

For cluster administrators

- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure Default CPU Requests and Limits for a Namespace

This page shows how to configure default CPU requests and limits for a namespace. A Kubernetes cluster can be divided into namespaces. If a Container is created in a namespace that has a default CPU limit, and the Container does not specify its own CPU limit, then the Container is assigned the default CPU limit. Kubernetes assigns a default CPU request under certain conditions that are explained later in this topic.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace default-cpu-example
```

Create a LimitRange and a Pod

Here's the configuration file for a LimitRange object. The configuration specifies a default CPU request and a default CPU limit.

cpu-defaults.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1
    defaultRequest:
      cpu: 0.5
    type: Container
```

Create the LimitRange in the default-cpu-example namespace:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults.yaml -
```

Now if a Container is created in the default-cpu-example namespace, and the Container does not specify its own values for CPU request and CPU limit, the Container is given a default CPU request of 0.5 and a default CPU limit of 1.

Here's the configuration file for a Pod that has one Container. The Container does not specify a CPU request and limit.

cpu-defaults-pod.yaml

cpu-defaults-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo
spec:
  containers:
  - name: default-cpu-demo-ctr
    image: nginx
```

Create the Pod.

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod.ya
```

View the Pod’s specification:

```
kubectl get pod default-cpu-demo --output=yaml --namespace=default-cpu-example
```

The output shows that the Pod’s Container has a CPU request of 500 millicpus and a CPU limit of 1 cpu. These are the default values specified by the LimitRange.

```
containers:
- image: nginx
  imagePullPolicy: Always
  name: default-cpu-demo-ctr
  resources:
    limits:
      cpu: "1"
    requests:
      cpu: 500m
```

What if you specify a Container’s limit, but not its request?

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU limit, but not a request:

cpu-defaults-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod-2.
```

View the Pod specification:

```
kubectl get pod cpu-limit-no-request --output=yaml --namespace=default-cpu-example
```

The output shows that the Container's CPU request is set to match its CPU limit. Notice that the Container was not assigned the default CPU request value of 0.5 cpu.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: "1"
```

What if you specify a Container's request, but not its limit?

Here’s the configuration file for a Pod that has one Container. The Container specifies a CPU request, but not a limit:

cpu-defaults-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-3
spec:
  containers:
  - name: default-cpu-demo-3-ctr
    image: nginx
    resources:
      requests:
        cpu: "0.75"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod-3.
```

The output shows that the Container’s CPU request is set to the value specified in the Container’s configuration file. The Container’s CPU limit is set to 1 cpu, which is the default CPU limit for the namespace.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 750m
```

Motivation for default CPU limits and requests

If your namespace has a resource quota, it is helpful to have a default value in place for CPU limit. Here are two of the restrictions that a resource quota imposes on a namespace:

- Every Container that runs in the namespace must have its own CPU limit.

- The total amount of CPU used by all Containers in the namespace must not exceed a specified limit.

If a Container does not specify its own CPU limit, it is given the default limit, and then it can be allowed to run in a namespace that is restricted by a quota.

What's next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure Minimum and Maximum Memory Constraints for a Namespace

This page shows how to set minimum and maximum values for memory used by Containers running in a namespace. You specify minimum and maximum memory values in a LimitRange object. If a Pod does not meet the constraints imposed by the LimitRange, it cannot be created in the namespace.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 1 GiB of memory.

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace constraints-mem-example
```

Create a LimitRange and a Pod

Here's the configuration file for a LimitRange:

memory-constraints.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

Create the LimitRange:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints.
```

View detailed information about the LimitRange:

```
kubectl get limitrange cpu-min-max-demo --namespace=constraints-mem-example --outp
```

The output shows the minimum and maximum memory constraints as expected. But notice that even though you didn't specify default values in the configuration file for the LimitRange, they were created automatically.

```
limits:
- default:
    memory: 1Gi
  defaultRequest:
    memory: 1Gi
  max:
    memory: 1Gi
  min:
    memory: 500Mi
  type: Container
```

Now whenever a Container is created in the constraints-mem-example namespace, Kubernetes performs these steps:

- If the Container does not specify its own memory request and limit, assign the default memory request and limit to the Container.
- Verify that the Container has a memory request that is greater than or equal to 500 MiB.
- Verify that the Container has a memory limit that is less than or equal to 1 GiB.

Here's the configuration file for a Pod that has one Container. The Container manifest specifies a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the minimum and maximum memory constraints imposed by the LimitRange.

memory-constraints-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo
spec:
  containers:
  - name: constraints-mem-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "600Mi"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-
```

Verify that the Pod's Container is running:

```
kubectl get pod constraints-mem-demo --namespace=constraints-mem-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo --output=yaml --namespace=constraints-mem-exa
```

The output shows that the Container has a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the constraints imposed by the LimitRange.

```
resources:
  limits:
     memory: 800Mi
  requests:
    memory: 600Mi
```

Delete your Pod:

```
kubectl delete pod constraints-mem-demo --namespace=constraints-mem-example
```

Attempt to create a Pod that exceeds the maximum memory constraint

Here's the configuration file for a Pod that has one Container. The Container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB.

memory-constraints-pod-2.yaml

memory-constraints-pod-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-2
spec:
  containers:
  - name: constraints-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1.5Gi"
      requests:
        memory: "800Mi"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-
```

The output shows that the Pod does not get created, because the Container specifies a memory limit that is too large:

```
Error from server (Forbidden): error when creating "docs/tasks/administer-cluster/
pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Container is
```

Attempt to create a Pod that does not meet the minimum memory request

Here's the configuration file for a Pod that has one Container. The Container specifies a memory request of 200 MiB and a memory limit of 800 MiB.

memory-constraints-pod-3.yaml

memory-constraints-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-3
spec:
  containers:
  - name: constraints-mem-demo-3-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "100Mi"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-
```

The output shows that the Pod does not get created, because the Container specifies a memory request that is too small:

```
Error from server (Forbidden): error when creating "docs/tasks/administer-cluster/
pods "constraints-mem-demo-3" is forbidden: minimum memory usage per Container is
```

Create a Pod that does not specify any memory request or limit

Here's the configuration file for a Pod that has one Container. The Container does not specify a memory request, and it does not specify a memory limit.

memory-constraints-pod-4.yaml

memory-constraints-pod-4.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-4
spec:
  containers:
  - name: constraints-mem-demo-4-ctr
    image: nginx
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo-4 --namespace=constraints-mem-example --outpu
```

The output shows that the Pod's Container has a memory request of 1 GiB and a memory limit of 1 GiB. How did the Container get those values?

```
resources:
  limits:
    memory: 1Gi
  requests:
    memory: 1Gi
```

Because your Container did not specify its own memory request and limit, it was given the default memory request and limit from the LimitRange.

At this point, your Container might be running or it might not be running. Recall that a prerequisite for this task is that your Nodes have at least 1 GiB of memory. If each of your Nodes has only 1 GiB of memory, then there is not enough allocatable memory on any Node to accommodate a memory request of 1 GiB. If you happen to be using Nodes with 2 GiB of memory, then you probably have enough space to accommodate the 1 GiB request.

Delete your Pod:

```
kubectl delete pod constraints-mem-demo-4 --namespace=constraints-mem-example
```

Enforcement of minimum and maximum memory constraints

The maximum and minimum memory constraints imposed on a namespace by a LimitRange are enforced only when a Pod is created or updated. If you change the LimitRange, it does not affect Pods that were created previously.

Motivation for minimum and maximum memory constraints

As a cluster administrator, you might want to impose restrictions on the amount of memory that Pods can use. For example:

- Each Node in a cluster has 2 GB of memory. You do not want to accept any Pod that requests more than 2 GB of memory, because no Node in the cluster can support the request.
- A cluster is shared by your production and development departments. You want to allow production workloads to consume up to 8 GB of memory, but you want development workloads to be limited to 512 MB. You create separate namespaces for production and development, and you apply memory constraints to each namespace.

Clean up

Delete your namespace:

```
kubectl delete namespace constraints-mem-example
```

What’s next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure Minimum and Maximum CPU Constraints for a Namespace

This page shows how to set minimum and maximum values for the CPU resources used by Containers and Pods in a namespace. You specify minimum and maximum CPU values in a LimitRange object. If a Pod does not meet the constraints imposed by the LimitRange, it cannot be created in the namespace.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 1 CPU.

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace constraints-cpu-example
```

Create a LimitRange and a Pod

Here's the configuration file for a LimitRange:

cpu-constraints.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-min-max-demo-lr
spec:
  limits:
  - max:
      cpu: "800m"
    min:
      cpu: "200m"
    type: Container
```

Create the LimitRange:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints.yam
```

View detailed information about the LimitRange:

```
kubectl get limitrange cpu-min-max-demo-lr --output=yaml --namespace=constraints-c
```

The output shows the minimum and maximum CPU constraints as expected. But notice that even though you didn't specify default values in the configuration file for the LimitRange, they were created automatically.

```
limits:
- default:
    cpu: 800m
  defaultRequest:
    cpu: 800m
  max:
    cpu: 800m
  min:
    cpu: 200m
  type: Container
```

Now whenever a Container is created in the constraints-cpu-example namespace, Kubernetes performs these steps:

- If the Container does not specify its own CPU request and limit, assign the default CPU request and limit to the Container.
- Verify that the Container specifies a CPU request that is greater than or equal to 200 millicpu.
- Verify that the Container specifies a CPU limit that is less than or equal to 800 millicpu.

Here's the configuration file for a Pod that has one Container. The Container manifest specifies a CPU request of 500 millicpu and a CPU limit of 800 millicpu. These satisfy the minimum and maximum CPU constraints imposed by the LimitRange.

cpu-constraints-pod.yaml

cpu-constraints-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo
spec:
  containers:
  - name: constraints-cpu-demo-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "500m"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod
```

Verify that the Pod's Container is running:

```
kubectl get pod constraints-cpu-demo --namespace=constraints-cpu-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo --output=yaml --namespace=constraints-cpu-exa
```

The output shows that the Container has a CPU request of 500 millicpu and CPU limit of 800 millicpu. These satisfy the constraints imposed by the LimitRange.

```
resources:
  limits:
    cpu: 800m
  requests:
    cpu: 500m
```

Delete the Pod

```
kubectl delete pod constraints-cpu-demo --namespace=constraints-cpu-example
```

Attempt to create a Pod that exceeds the maximum CPU constraint

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU request of 500 millicpu and a cpu limit of 1.5 cpu.

cpu-constraints-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-2
spec:
  containers:
  - name: constraints-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1.5"
      requests:
        cpu: "500m"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod
```

The output shows that the Pod does not get created, because the Container specifies a CPU limit that is too large:

```
Error from server (Forbidden): error when creating "docs/tasks/administer-cluster/
pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800
```

Attempt to create a Pod that does not meet the minimum CPU request

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU request of 100 millicpu and a CPU limit of 800 millicpu.

cpu-constraints-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod
```

The output shows that the Pod does not get created, because the Container specifies a CPU request that is too small:

```
Error from server (Forbidden): error when creating "docs/tasks/administer-cluster/
pods "constraints-cpu-demo-4" is forbidden: minimum cpu usage per Container is 200
```

Create a Pod that does not specify any CPU request or limit

Here's the configuration file for a Pod that has one Container. The Container does not specify a CPU request, and it does not specify a CPU limit.

cpu-constraints-pod-4.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: vish/stress
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo-4 --namespace=constraints-cpu-example --outpu
```

The output shows that the Pod's Container has a CPU request of 800 millicpu and a CPU limit of 800 millicpu. How did the Container get those values?

```
resources:
  limits:
    cpu: 800m
  requests:
    cpu: 800m
```

Because your Container did not specify its own CPU request and limit, it was given the default CPU request and limit from the LimitRange.

At this point, your Container might be running or it might not be running. Recall that a prerequisite for this task is that your Nodes have at least 1 CPU. If each of your Nodes has only 1 CPU, then there might not be enough allocatable CPU on any Node to accommodate a request of 800 millicpu. If you happen to be using Nodes with 2 CPU, then you probably have enough CPU to accommodate the 800 millicpu request.

Delete your Pod:

```
kubectl delete pod constraints-cpu-demo-4 --namespace=constraints-cpu-example
```

Enforcement of minimum and maximum CPU constraints

The maximum and minimum CPU constraints imposed on a namespace by a LimitRange are enforced only when a Pod is created or updated. If you change the LimitRange, it does not affect Pods that were created previously.

Motivation for minimum and maximum CPU constraints

As a cluster administrator, you might want to impose restrictions on the CPU resources that Pods can use. For example:

- Each Node in a cluster has 2 CPU. You do not want to accept any Pod that requests more than 2 CPU, because no Node in the cluster can support the request.
- A cluster is shared by your production and development departments. You want to allow production workloads to consume up to 3 CPU, but you want development workloads to be limited to 1 CPU. You create separate namespaces for production and development, and you apply CPU constraints to each namespace.

Clean up

Delete your namespace:

```
kubectl delete namespace constraints-cpu-example
```

What’s next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure Memory and CPU Quotas for a Namespace

This page shows how to set quotas for the total amount memory and CPU that can be used by all Containers running in a namespace. You specify quotas in a ResourceQuota object.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Each node in your cluster must have at least 1 GiB of memory.

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-mem-cpu-example
```

Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

quota-mem-cpu.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Create the ResourceQuota:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-mem-cpu.yaml
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=
```

The ResourceQuota places these requirements on the quota-mem-cpu-example namespace:

- Every Container must have a memory request, memory limit, cpu request, and cpu limit.
- The memory request total for all Containers must not exceed 1 GiB.
- The memory limit total for all Containers must not exceed 2 GiB.
- The CPU request total for all Containers must not exceed 1 cpu.

- The CPU limit total for all Containers must not exceed 2 cpu.

Create a Pod

Here is the configuration file for a Pod:

quota-mem-cpu-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo
spec:
  containers:
  - name: quota-mem-cpu-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
        cpu: "800m"
      requests:
        memory: "600Mi"
        cpu: "400m"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-mem-cpu-pod.y
```

Verify that the Pod's Container is running:

```
kubectl get pod quota-mem-cpu-demo --namespace=quota-mem-cpu-example
```

Once again, view detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=
```

The output shows the quota along with how much of the quota has been used. You can see that the memory and CPU requests and limits for your Pod do not exceed the quota.

```
status:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.cpu: "1"
    requests.memory: 1Gi
  used:
    limits.cpu: 800m
    limits.memory: 800Mi
    requests.cpu: 400m
    requests.memory: 600Mi
```

Attempt to create a second Pod

Here is the configuration file for a second Pod:

quota-mem-cpu-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo-2
spec:
  containers:
  - name: quota-mem-cpu-demo-2-ctr
    image: redis
    resources:
      limits:
        memory: "1Gi"
        cpu: "800m"
      requests:
        memory: "700Mi"
        cpu: "400m"
```

In the configuration file, you can see that the Pod has a memory request of 700 MiB. Notice that the sum of the used memory request and this new memory request exceeds the memory request quota. 600 MiB + 700 MiB > 1 GiB.

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-mem-cpu-pod-2
```

The second Pod does not get created. The output shows that creating the second Pod would cause the memory request total to exceed the memory request quota.

```
Error from server (Forbidden): error when creating "docs/tasks/administer-cluster/
pods "quota-mem-cpu-demo-2" is forbidden: exceeded quota: mem-cpu-demo,
requested: requests.memory=700Mi,used: requests.memory=600Mi, limited: requests.me
```

Discussion

As you have seen in this exercise, you can use a ResourceQuota to restrict the memory request total for all Containers running in a namespace. You can also restrict the totals for memory limit, cpu request, and cpu limit.

If you want to restrict individual Containers, instead of totals for all Containers, use a LimitRange.

Clean up

Delete your namespace:

```
kubectl delete namespace quota-mem-cpu-example
```

What's next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace

- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure a Pod Quota for a Namespace

This page shows how to set a quota for the total number of Pods that can run in a namespace. You specify quotas in a ResourceQuota object.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-pod-example
```

Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

quota-pod.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "2"
```

Create the ResourceQuota:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-pod.yaml --na
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota pod-demo --namespace=quota-pod-example --output=yaml
```

The output shows that the namespace has a quota of two Pods, and that currently there are no Pods; that is, none of the quota is used.

```
spec:
  hard:
    pods: "2"
status:
  hard:
    pods: "2"
  used:
    pods: "0"
```

Here is the configuration file for a Deployment:

quota-pod-deployment.yaml

quota-pod-deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: pod-quota-demo
spec:
  replicas: 3
  template:
    metadata:
      labels:
        purpose: quota-demo
    spec:
      containers:
      - name: pod-quota-demo
        image: nginx
```

In the configuration file, `replicas: 3` tells Kubernetes to attempt to create three Pods, all running the same application.

Create the Deployment:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-pod-deploymen
```

View detailed information about the Deployment:

```
kubectl get deployment pod-quota-demo --namespace=quota-pod-example --output=yaml
```

The output shows that even though the Deployment specifies three replicas, only two Pods were created because of the quota.

```
spec:
  ...
  replicas: 3
...
status:
  availableReplicas: 2
...
lastUpdateTime: 2017-07-07T20:57:05Z
    message: 'unable to create pods: pods "pod-quota-demo-1650323038-" is forbidde
      exceeded quota: pod-demo, requested: pods=1, used: pods=2, limited: pods=2'
```

Clean up

Delete your namespace:

```
kubectl delete namespace quota-pod-example
```

What's next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure Quotas for API Objects

For app developers

- Assign Memory Resources to Containers and Pods

- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Configure Quotas for API Objects

This page shows how to configure quotas for API objects, including PersistentVolumeClaims and Services. A quota restricts the number of objects, of a particular type, that can be created in a namespace. You specify quotas in a ResourceQuota object.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-object-example
```

Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

quota-objects.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-quota-demo
spec:
  hard:
    persistentvolumeclaims: "1"
    services.loadbalancers: "2"
    services.nodeports: "0"
```

Create the ResourceQuota:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects.yaml
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota object-quota-demo --namespace=quota-object-example --out
```

The output shows that in the quota-object-example namespace, there can be at most one PersistentVolumeClaim, at most two Services of type LoadBalancer, and no Services of type NodePort.

```
status:
  hard:
    persistentvolumeclaims: "1"
    services.loadbalancers: "2"
    services.nodeports: "0"
  used:
    persistentvolumeclaims: "0"
    services.loadbalancers: "0"
    services.nodeports: "0"
```

Create a PersistentVolumeClaim:

Here is the configuration file for a PersistentVolumeClaim object:

quota-objects-pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-quota-demo
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects-pvc.y
```

Verify that the PersistentVolumeClaim was created:

```
kubectl get persistentvolumeclaims --namespace=quota-object-example
```

The output shows that the PersistentVolumeClaim exists and has status Pending:

```
NAME             STATUS
pvc-quota-demo   Pending
```

Attempt to create a second PersistentVolumeClaim:

Here is the configuration file for a second PersistentVolumeClaim:

quota-objects-pvc-2.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-quota-demo-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
```

Attempt to create the second PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects-pvc-2
```

The output shows that the second PersistentVolumeClaim was not created, because it would have exceeded the quota for the namespace.

```
persistentvolumeclaims "pvc-quota-demo-2" is forbidden:
exceeded quota: object-quota-demo, requested: persistentvolumeclaims=1,
used: persistentvolumeclaims=1, limited: persistentvolumeclaims=1
```

Notes

These are the strings used to identify API resources that can be constrained by quotas:

String	API Object
"pods"	Pod
"services	Service
"replicationcontrollers"	ReplicationController
"resourcequotas"	ResourceQuota
"secrets"	Secret
"configmaps"	ConfigMap
"persistentvolumeclaims"	PersistentVolumeClaim
"services.nodeports"	Service of type NodePort
"services.loadbalancers"	Service of type LoadBalancer

Clean up

Delete your namespace:

```
kubectl delete namespace quota-object-example
```

What's next

For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace

For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

Advertise Opaque Integer Resources for a Node

This page shows how to specify opaque integer resources for a Node. Opaque integer resources allow cluster administrators to advertise node-level resources that would otherwise be unknown to Kubernetes.

DEPRECATION NOTICE: As of `Kubernetes v1.8`, this has been deprecated

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Get the names of your Nodes

```
kubectl get nodes
```

Choose one of your Nodes to use for this exercise.

Advertise a new opaque integer resource on one of your Nodes

To advertise a new opaque integer resource on a Node, send an HTTP PATCH request to the Kubernetes API server. For example, suppose one of your Nodes has four dongles attached. Here's an example of a PATCH request that advertises four dongle resources for your Node.

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "add",
    "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-int-resource-dongle"
    "value": "4"
  }
]
```

Note that Kubernetes does not need to know what a dongle is or what a dongle is for. The preceding PATCH request just tells Kubernetes that your Node has four things that you call dongles.

Start a proxy, so that you can easily send requests to the Kubernetes API server:

```
kubectl proxy
```

In another command window, send the HTTP PATCH request. Replace `<your-node-name>` with the name of your Node:

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-i
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

Note: In the preceding request, `~1` is the encoding for the character / in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

The output shows that the Node has a capacity of 4 dongles:

```
"capacity": {
  "alpha.kubernetes.io/nvidia-gpu": "0",
  "cpu": "2",
  "memory": "2049008Ki",
  "pod.alpha.kubernetes.io/opaque-int-resource-dongle": "4",
```

Describe your Node:

```
kubectl describe node <your-node-name>
```

Once again, the output shows the dongle resource:

```
Capacity:
 alpha.kubernetes.io/nvidia-gpu:      0
 cpu:                 2
 memory:                2049008Ki
 pod.alpha.kubernetes.io/opaque-int-resource-dongle:   4
```

Now, application developers can create Pods that request a certain number of dongles. See Assign Opaque Integer Resources to a Container.

Discussion

Opaque integer resources are similar to memory and CPU resources. For example, just as a Node has a certain amount of memory and CPU to be shared by all components running on the Node, it can have a certain number of dongles to be shared by all components running on the Node. And just as application developers can create Pods that request a certain amount of memory and CPU, they can create Pods that request a certain number of dongles.

Opaque integer resources are called opaque because Kubernetes does not know anything about what they are. Kubernetes knows only that a Node has a certain number of them. They are called integer resources because they must be advertised in integer amounts. For example, a Node can advertise four dongles, but not 4.5 dongles.

Storage example

Suppose a Node has 800 GiB of a special kind of disk storage. You could create a name for the special storage, say opaque-int-resource-special-storage. Then you could advertise it in chunks of a certain size, say 100 GiB. In that case, your Node would advertise that it has eight resources of type opaque-int-resource-special-storage.

```
Capacity:
 ...
 pod.alpha.kubernetes.io/opaque-int-resource-special-storage:  8
```

If you want to allow arbitrary requests for special storage, you could advertise special storage in chunks of size 1 byte. In that case, you would advertise 800Gi resources of type opaque-int-resource-special-storage.

```
Capacity:
 ...
 pod.alpha.kubernetes.io/opaque-int-resource-special-storage:  800Gi
```

Then a Container could request any number of bytes of special storage, up to 800Gi.

Clean up

Here is a PATCH request that removes the dongle advertisement from a Node.

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "remove",
    "path": "/status/capacity/pod.alpha.kubernetes.io~1opaque-int-resource-dongle"
  }
]
```

Start a proxy, so that you can easily send requests to the Kubernetes API server:

```
kubectl proxy
```

In another command window, send the HTTP PATCH request. Replace `<your-node-name>` with the name of your Node:

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "remove", "path": "/status/capacity/pod.alpha.kubernetes.io~1opaqu
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

Verify that the dongle advertisement has been removed:

```
kubectl describe node <your-node-name> | grep dongle
```

What's next

For application developers

- Assign Opaque Integer Resources to a Container

For cluster administrators

- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace

Control CPU Management Policies on the Node

Kubernetes keeps many aspects of how pods execute on nodes abstracted from the user. This is by design. However, some workloads require stronger guarantees in terms of latency and/or performance in order to operate acceptably. The kubelet provides methods to enable more complex workload placement policies while keeping the abstraction free from explicit placement directives.

CPU Management Policies

By default, the kubelet uses CFS quota to enforce pod CPU limits. When the node runs many CPU-bound pods, the workload can move to different CPU cores depending on whether the pod is throttled and which CPU cores are available at scheduling time. Many workloads are not sensitive to this migration and thus work fine without any intervention.

However, in workloads where CPU cache affinity and scheduling latency significantly affect workload performance, the kubelet allows alternative CPU management policies to determine some placement preferences on the node.

Configuration

The CPU Manager is introduced as an alpha feature in Kubernetes v1.8. It must be explicitly enabled in the kubelet feature gates: `--feature-gates=CPUManager=true` .

The CPU Manager policy is set with the `--cpu-manager-policy` kubelet option. There are two supported policies:

- `none` : the default, which represents the existing scheduling behavior.

- `static`: allows pods with certain resource characteristics to be granted increased CPU affinity and exclusivity on the node.

The CPU manager periodically writes resource updates through the CRI in order to reconcile in-memory CPU assignments with cgroupfs. The reconcile frequency is set through a new Kubelet configuration value `--cpu-manager-reconcile-period`. If not specified, it defaults to the same duration as `--node-status-update-frequency`.

None policy

The `none` policy explicitly enables the existing default CPU affinity scheme, providing no affinity beyond what the OS scheduler does automatically. Limits on CPU usage for Guaranteed pods are enforced using CFS quota.

Static policy

The `static` policy allows containers in `Guaranteed` pods with integer CPU `requests` access to exclusive CPUs on the node. This exclusivity is enforced using the cpuset cgroup controller.

Note: System services such as the container runtime and the kubelet itself can continue to run on these exclusive CPUs. The exclusivity only extends to other pods.

Note: The alpha version of this policy does not guarantee static exclusive allocations across Kubelet restarts.

This policy manages a shared pool of CPUs that initially contains all CPUs in the node. The amount of exclusively allocatable CPUs is equal to the total number of CPUs in the node minus any CPU reservations by the kubelet `--kube-reserved` or `--system-reserved` options. CPUs reserved by these options are taken, in integer quantity, from the initial shared pool in ascending order by physical core ID. This shared pool is the set of CPUs on which any containers in `BestEffort` and `Burstable` pods run. Containers in `Guaranteed` pods with fractional CPU `requests` also run on

CPUs in the shared pool. Only containers that are both part of a `Guaranteed` pod and have integer CPU `requests` are assigned exclusive CPUs.

> **Note:** The kubelet requires a CPU reservation greater than zero be made using either `--kube-reserved` and/or `--system-reserved` when the static policy is enabled. This is because zero CPU reservation would allow the shared pool to become empty.

As `Guaranteed` pods whose containers fit the requirements for being statically assigned are scheduled to the node, CPUs are removed from the shared pool and placed in the cpuset for the container. CFS quota is not used to bound the CPU usage of these containers as their usage is bound by the scheduling domain itself. In others words, the number of CPUs in the container cpuset is equal to the integer CPU `limit` specified in the pod spec. This static assignment increases CPU affinity and decreases context switches due to throttling for the CPU-bound workload.

Consider the containers in the following pod specs:

```
spec:
  containers:
  - name: nginx
    image: nginx
```

This pod runs in the `BestEffort` QoS class because no resource `requests` or `limits` are specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

This pod runs in the `Burstable` QoS class because resource `requests` do not equal `limits` and the `cpu` quantity is not specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "100Mi"
        cpu: "1"
```

This pod runs in the `Burstable` QoS class because resource `requests` do not equal `limits`. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "200Mi"
        cpu: "2"
```

This pod runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. And the container's resource limit for the CPU resource is an integer greater than or equal to one. The `nginx` container is granted 2 exclusive CPUs.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "1.5"
      requests:
        memory: "200Mi"
        cpu: "1.5"
```

This pod runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. But the container's resource limit for the CPU resource is a fraction. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
```

This pod runs in the `Guaranteed` QoS class because only `limits` are specified and `requests` are set equal to `limits` when not explicitly specified. And the container's resource limit for the CPU resource is an integer greater than or equal to one.The `nginx` container is granted 2 exclusive CPUs.

Access Clusters Using the Kubernetes API

This page shows how to access clusters using the Kubernetes API.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Accessing the cluster API

Accessing for the first time with kubectl

When accessing the Kubernetes API for the first time, use the Kubernetes command-line tool, `kubectl`.

To access a cluster, you need to know the location of the cluster and have credentials to access it. Typically, this is automatically set-up when you work through a Getting started guide, or someone else setup the cluster and provided you with credentials and a location.

Check the location and credentials that kubectl knows about with this command:

```
$ kubectl config view
```

Many of the examples provide an introduction to using kubectl. Complete documentation is found in the kubectl manual.

Directly accessing the REST API

Kubectl handles locating and authenticating to the apiserver. If you want to directly access the REST API with an http client like `curl` or `wget`, or a browser, there are multiple ways you can locate and authenticate against the apiserver:

1. Run kubectl in proxy mode (recommended). This method is recommended, since it uses the stored apiserver location abd verifies the identity of the apiserver using a self-signed cert. No Man-in-the-middle (MITM) attack is possible using this method .
2. Alternatively, you can provide the location and credentials directly to the http client. This works with for client code that is confused by proxies. To protect against man in the middle attacks, you'll need to import a root cert into your browser.

Using the Go or Python client libraries provides accessing kubectl in proxy mode.

Using kubectl proxy

The following command runs kubectl in a mode where it acts as a reverse proxy. It handles locating the apiserver and authenticating.

Run it like this:

```
$ kubectl proxy --port=8080 &
```

See kubectl proxy for more details.

Then you can explore the API with curl, wget, or a browser, like so:

```
$ curl http://localhost:8080/api/
{
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

Without kubectl proxy

It is possible to avoid using kubectl proxy by passing an authentication token directly to the apiserver, like this:

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

The above example uses the `--insecure` flag. This leaves it subject to MITM attacks. When kubectl accesses the cluster it uses a stored root certificate and client certificates to access the server. (These are installed in the `~/.kube` directory). Since cluster certificates are typically self-signed, it may take special configuration to get your http client to use root certificate.

On some clusters, the apiserver does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. Configuring Access to the API describes how a cluster admin can configure this. Such approaches may conflict with future high-availability support.

Programmatic access to the API

Kubernetes officially supports client libraries for Go and Python.

Go client

- To get the library, run the following command: `go get k8s.io/client-go/<version number>/kubernetes` See https://github.com/kubernetes/client-go to see which versions are supported.
- Write an application atop of the client-go clients. Note that client-go defines its own API objects, so if needed, please import API definitions from client-go rather than from the main repository, e.g., `import "k8s.io/client-go/1.4/pkg/api/v1"` is correct.

The Go client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the apiserver. See this example:

```
import (
   "fmt"
   "k8s.io/client-go/1.4/kubernetes"
   "k8s.io/client-go/1.4/pkg/api/v1"
   "k8s.io/client-go/1.4/tools/clientcmd"
)
...
   // uses the current context in kubeconfig
   config, _ := clientcmd.BuildConfigFromFlags("", "path to kubeconfig")
   // creates the clientset
   clientset, _:= kubernetes.NewForConfig(config)
   // access the API to list pods
   pods, _:= clientset.Core().Pods("").List(v1.ListOptions{})
   fmt.Printf("There are %d pods in the cluster\n", len(pods.Items))
...
```

If the application is deployed as a Pod in the cluster, please refer to the next section.

Python client

To use Python client, run the following command: `pip install kubernetes` See Python Client Library page for more installation options.

The Python client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the apiserver. See this example:

```
from kubernetes import client, config

config.load_kube_config()

v1=client.CoreV1Api()
print("Listing pods with their IPs:")
ret = v1.list_pod_for_all_namespaces(watch=False)
for i in ret.items:
    print("%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
```

Other languages

There are client libraries for accessing the API from other languages. See documentation for other libraries for how they authenticate.

Accessing the API from a Pod

When accessing the API from a pod, locating and authenticating to the API server are somewhat different.

The recommended way to locate the apiserver within the pod is with the `kubernetes` DNS name, which resolves to a Service IP which in turn will be routed to an apiserver.

The recommended way to authenticate to the apiserver is with a service account credential. By kube-system, a pod is associated with a service account, and a credential (token) for that service account is placed into the filesystem tree of each container in that pod, at `/var/run/secrets/kubernetes.io/serviceaccount/token` .

If available, a certificate bundle is placed into the filesystem tree of each container at `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` , and should be used to verify the serving certificate of the apiserver.

Finally, the default namespace to be used for namespaced API operations is placed in a file at `/var/run/secrets/kubernetes.io/serviceaccount/namespace` in each container.

From within a pod the recommended ways to connect to API are:

- run a kubectl proxy as one of the containers in the pod, or as a background process within a container. This proxies the Kubernetes API to the localhost interface of the pod, so that other processes in any container of the pod can access it. See this example of using kubectl proxy in a pod.
- use the Go client library, and create a client using the `rest.InClusterConfig()` and `kubernetes.NewForConfig()` functions. They handle locating and authenticating to the apiserver. example

In each case, the credentials of the pod are used to communicate securely with the apiserver.

Access Services Running on Clusters

This page shows how to connect to services running on the Kubernetes cluster.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Accessing services running on the cluster

In Kubernetes, nodes, pods and services all have their own IPs. In many cases, the node IPs, pod IPs, and some service IPs on a cluster will not be routable, so they will not be reachable from a machine outside the cluster, such as your desktop machine.

Ways to connect

You have several options for connecting to nodes, pods and services from outside the cluster:

- Access services through public IPs.

  - Use a service with type `NodePort` or `LoadBalancer` to make the service reachable outside the cluster. See the services and kubectl expose documentation.
  - Depending on your cluster environment, this may just expose the service to your corporate network, or it may expose it to the internet. Think about whether the service being exposed is secure. Does it do its own authentication?
  - Place pods behind services. To access one specific pod from a set of replicas, such as for debugging, place a unique label on the pod it and create a new service which selects this label.
  - In most cases, it should not be necessary for application developer to directly access nodes via their nodeIPs.
- Access services, nodes, or pods using the Proxy Verb.
  - Does apiserver authentication and authorization prior to accessing the remote service. Use this if the services are not secure enough to expose to the internet, or to gain access to ports on the node IP, or for debugging.
  - Proxies may cause problems for some web applications.
  - Only works for HTTP/HTTPS.
  - Described here.
- Access from a node or pod in the cluster.
  - Run a pod, and then connect to a shell in it using kubectl exec. Connect to other nodes, pods, and services from that shell.
  - Some clusters may allow you to ssh to a node in the cluster. From there you may be able to access cluster services. This is a non-standard method, and will work on some clusters but not others. Browsers and other tools may or may not be installed. Cluster DNS may not work.

Discovering builtin services

Typically, there are several services which are started on a cluster by kube-system. Get a list of these with the `kubectl cluster-info` command:

```
$ kubectl cluster-info

  Kubernetes master is running at https://104.197.5.247
  elasticsearch-logging is running at https://104.197.5.247/api/v1/namespaces/kube
  kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system
  kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/servi
  grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/servic
  heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/servi
```

This shows the proxy-verb URL for accessing each service. For example, this cluster has cluster-level logging enabled (using Elasticsearch), which can be reached at `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` if suitable credentials are passed, or through a kubectl proxy at, for example: `http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`. (See above for how to pass credentials or use kubectl proxy.)

Manually constructing apiserver proxy URLs

As mentioned above, you use the `kubectl cluster-info` command to retrieve the service's proxy URL. To create proxy URLs that include service endpoints, suffixes, and parameters, you simply append to the service's proxy URL: `http://` *`kubernetes_master_address`* `/api/v1/namespaces/` *`namespace_name`* `/services/` *`service_name[:port_name]`* `/proxy`

If you haven't specified a name for your port, you don't have to specify *port_name* in the URL

Examples

- To access the Elasticsearch service endpoint `_search?q=user:kimchy`, you would use: `http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy`
- To access the Elasticsearch cluster health information `_cluster/health?pretty=true`, you would use:

```
https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-
logging/proxy/_cluster/health?pretty=true
```

```
{
  "cluster_name" : "kubernetes_logging",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

Using web browsers to access services running on the cluster

You may be able to put an apiserver proxy URL into the address bar of a browser. However:

- Web browsers cannot usually pass tokens, so you may need to use basic (password) auth. Apiserver can be configured to accept basic auth, but your cluster may not be configured to accept basic auth.
- Some web apps may not work, particularly those with client side javascript that construct URLs in a way that is unaware of the proxy path prefix.

Securing a Cluster

This document covers topics related to protecting a cluster from accidental or malicious access and provides recommendations on overall security.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda

- Play with Kubernetes

Controlling access to the Kubernetes API

As Kubernetes is entirely API driven, controlling and limiting who can access the cluster and what actions they are allowed to perform is the first line of defense.

Use Transport Level Security (TLS) for all API traffic

Kubernetes expects that all API communication in the cluster is encrypted by default with TLS, and the majority of installation methods will allow the necessary certificates to be created and distributed to the cluster components. Note that some components and installation methods may enable local ports over HTTP and administrators should familiarize themselves with the settings of each component to identify potentially unsecured traffic.

API Authentication

Choose an authentication mechanism for the API servers to use that matches the common access patterns when you install a cluster. For instance, small single user clusters may wish to use a simple certificate or static Bearer token approach. Larger clusters may wish to integrate an existing or OIDC or LDAP server that allow users to be subdivided into groups.

All API clients must be authenticated, even those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins. These clients are typically service accounts or use x509 client certificates, and they are created automatically at cluster startup or are setup as part of the cluster installation.

Consult the authentication reference document for more information.

API Authorization

Once authenticated, every API call is also expected to pass an authorization check. Kubernetes ships an integrated Role-Based Access Control (RBAC) component that matches an incoming user or group to a set of permissions bundled into roles. These permissions combine verbs (get, create, delete) with resources (pods, services, nodes) and can be namespace or cluster scoped. A set of out of the box roles are provided that offer reasonable default separation of responsibility depending on what actions a client might want to perform. It is recommended that you use the Node and RBAC authorizers together, in combination with the NodeRestriction admission plugin.

As with authentication, simple and broad roles may be appropriate for smaller clusters, but as more users interact with the cluster, it may become necessary to separate teams into separate namespaces with more limited roles.

With authorization, it is important to understand how updates on one object may cause actions in other places. For instance, a user may not be able to create pods directly, but allowing them to create a deployment, which creates pods on their behalf, will let them create those pods indirectly. Likewise, deleting a node from the API will result in the pods scheduled to that node being terminated and recreated on other nodes. The out of the box roles represent a balance between flexibility and the common use cases, but more limited roles should be carefully reviewed to prevent accidental escalation. You can make roles specific to your use case if the out-of-box ones don't meet your needs.

Consult the authorization reference section for more information.

Controlling the capabilities of a workload or user at runtime

Authorization in Kubernetes is intentionally high level, focused on coarse actions on resources. More powerful controls exist as **policies** to limit by use case how those objects act on the cluster, themselves, and other resources.

Limiting resource usage on a cluster

Resource quota limits the number or capacity of resources granted to a namespace. This is most often used to limit the amount of CPU, memory, or persistent disk a namespace can allocate, but can also control how many pods, services, or volumes exist in each namespace.

Limit ranges restrict the maximum or minimum size of some of the resources above, to prevent users from requesting unreasonably high or low values for commonly reserved resources like memory, or to provide default limits when none are specified.

Controlling what privileges containers run with

A pod definition contains a security context that allows it to request access to running as a specific Linux user on a node (like root), access to run privileged or access the host network, and other controls that would otherwise allow it to run unfettered on a hosting node. Pod security policies can limit which users or service accounts can provide dangerous security context settings. For example, pod security policies can limit volume mounts, especially `hostPath`, which are aspects of a pod that should be controlled.

Generally, most application workloads need limited access to host resources so they can successfully run as a root process (uid 0) without access to host information. However, considering the privileges associated with the root user, you should write application containers to run as a non-root user. Similarly, administrators who wish to prevent client applications from escaping their containers should use a restrictive pod security policy.

Restricting network access

The network policies for a namespace allows application authors to restrict which pods in other namespaces may access pods and ports within their namespaces. Many of the supported Kubernetes networking providers now respect network policy.

Quota and limit ranges can also be used to control whether users may request node ports or load balanced services, which on many clusters can control whether those users applications are visible outside of the cluster.

Additional protections may be available that control network rules on a per plugin or per environment basis, such as per-node firewalls, physically separating cluster nodes to prevent cross talk, or advanced networking policy.

Controlling which nodes pods may access

By default, there are no restrictions on which nodes may run a pod. Kubernetes offers a rich set of policies for controlling placement of pods onto nodes and the taint based pod placement and eviction that are available to end users. For many clusters use of these policies to separate workloads can be a convention that authors adopt or enforce via tooling.

As an administrator, a beta admission plugin `PodNodeSelector` can be used to force pods within a namespace to default or require a specific node selector, and if end users cannot alter namespaces, this can strongly limit the placement of all of the pods in a specific workload.

Protecting cluster components from compromise

This section describes some common patterns for protecting clusters from compromise.

Restrict access to etcd

Write access to the etcd backend for the API is equivalent to gaining root on the entire cluster, and read access can be used to escalate fairly quickly. Administrators should always use strong credentials from the API servers to their etcd server, such as mutual auth via TLS client certificates, and it is often recommended to isolate the etcd servers behind a firewall that only the API servers may access.

CAUTION: Allowing other components within the cluster to access the master etcd instance with read or write access to the full keyspace is equivalent to granting cluster-admin access. Using separate etcd instances for non-master components or using etcd ACLs to restrict read and write access to a subset of the keyspace is strongly recommended.

Enable audit logging

The audit logger is an alpha feature that records actions taken by the API for later analysis in the event of a compromise. It is recommended to enable audit logging and archive the audit file on a secure server.

Restrict access to alpha or beta features

Alpha and beta Kubernetes features are in active development and may have limitations or bugs that result in security vulnerabilities. Always assess the value an alpha or beta feature may provide against the possible risk to your security posture. When in doubt, disable features you do not use.

Rotate infrastructure credentials frequently

The shorter the lifetime of a secret or credential the harder it is for an attacker to make use of that credential. Set short lifetimes on certificates and automate their rotation. Use an authentication provider that can control how long issued tokens are available and use short lifetimes where possible. If you use service account tokens in external integrations, plan to rotate those tokens frequently. For example, once the bootstrap phase is complete, a bootstrap token used for setting up nodes should be revoked or its authorization removed.

Review third party integrations before enabling them

Many third party integrations to Kubernetes may alter the security profile of your cluster. When enabling an integration, always review the permissions that an extension requests before granting it access. For example, many security integrations may request access to view all secrets on your cluster which is effectively making that component a cluster admin. When in doubt, restrict the integration to functioning in a single namespace if possible.

Components that create pods may also be unexpectedly powerful if they can do so inside namespaces like the `kube-system` namespace, because those pods can gain access to service account secrets or run with elevated permissions if those service accounts are granted access to permissive pod security policies.

Encrypt secrets at rest

In general, the etcd database will contain any information accessible via the Kubernetes API and may grant an attacker significant visibility into the state of your cluster. Always encrypt your backups using a well reviewed backup and encryption solution, and consider using full disk encryption where possible.

Kubernetes 1.7 contains encryption at rest, an alpha feature that will encrypt `Secret` resources in etcd, preventing parties that gain access to your etcd backups from viewing the content of those secrets. While this feature is currently experimental, it may offer an additional level of defense when backups are not encrypted or an attacker gains read access to etcd.

Receiving alerts for security updates and reporting vulnerabilities

Join the kubernetes-announce group for emails about security announcements. See the security reporting page for more on how to report vulnerabilities.

Encrypting data at rest

This page shows how to enable and configure encryption of secret data at rest.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda

- Play with Kubernetes

- Kubernetes version 1.7.0 or later is required

- etcd v3 or later is required

- Encryption at rest is alpha in 1.7.0 which means it may change without notice. Users may be required to decrypt their data prior to upgrading to 1.8.0.

Configuration and determining whether encryption at rest is already enabled

The `kube-apiserver` process accepts an argument `--experimental-encryption-provider-config` that controls how API data is encrypted in etcd. An example configuration is provided below.

Understanding the encryption at rest configuration.

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - secretbox:
        keys:
        - name: key1
          secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
```

Each `resources` array item is a separate config and contains a complete configuration. The `resources.resources` field is an array of Kubernetes resource names (`resource` or `resource.group`) that should be encrypted. The `providers` array is an ordered list of the possible encryption providers. Only one provider type may be specified per entry (`identity` or `aescbc` may be provided, but not both in the same item).

The first provider in the list is used to encrypt resources going into storage. When reading resources from storage each provider that matches the stored data attempts to decrypt the data in order. If no provider can read the stored data due to a mismatch in format or secret key, an error is returned which prevents clients from accessing that resource.

IMPORTANT: If any resource is not readable via the encryption config (because keys were changed), the only recourse is to delete that key from the underlying etcd directly. Calls that attempt to read that resource will fail until it is deleted or a valid decryption key is provided.

Providers:

Name	Encryption	Strength	Speed	Key Length	Other Considerations
`identity`	None	N/A	N/A	N/A	Resources written as-is without encryption. When set as the first provider, the resource will be decrypted as new values are written.
`aescbc`	AES-CBC with PKCS#7 padding	Strongest	Fast	32-byte	The recommended choice for encryption at rest but may be slightly slower than `secretbox`.
`secretbox`	XSalsa20 and Poly1305	Strong	Faster	32-byte	A newer standard and may not be considered acceptable in environments that require high levels of review.
`aesgcm`	AES-GCM with random nonce	Must be rotated every 200k writes	Fastest	16, 24, or 32-byte	Is not recommended for use except when an automated key rotation scheme is implemented.

Each provider supports multiple keys - the keys are tried in order for decryption, and if the provider is the first provider, the first key is used for encryption.

Encrypting your data

Create a new encryption config file:

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - aescbc:
        keys:
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
    - identity: {}
```

To create a new secret perform the following steps:

1. Generate a 32 byte random key and base64 encode it. If you're on Linux or Mac OS X, run the following command:

   ```
   head -c 32 /dev/urandom | base64
   ```

2. Place that value in the secret field.
3. Set the `--experimental-encryption-provider-config` flag on the `kube-apiserver` to point to the location of the config file.
4. Restart your API server.

IMPORTANT: Your config file contains keys that can decrypt content in etcd, so you must properly restrict permissions on your masters so only the user who runs the kube-apiserver can read it.

Verifying that data is encrypted

Data is encrypted when written to etcd. After restarting your `kube-apiserver`, any newly created or updated secret should be encrypted when stored. To check, you can use the `etcdctl` command line program to retrieve the contents of your secret.

1. Create a new secret called `secret1` in the `default` namespace:

   ```
   kubectl create secret generic secret1 -n default --from-literal=mykey=mydata
   ```

2. Using the etcdctl commandline, read that secret out of etcd:

   ```
   ETCDCTL_API=3 etcdctl get /kubernetes.io/secrets/default/secret1 [...] |
   hexdump -C
   ```

   where `[...]` must be the additional arguments for connecting to the etcd server.

3. Verify the stored secret is prefixed with `k8s:enc:aescbc:v1:` which indicates the `aescbc` provider has encrypted the resulting data.
4. Verify the secret is correctly decrypted when retrieved via the API:

   ```
   kubectl describe secret secret1 -n default
   ```

should match `mykey: mydata`

Ensure all secrets are encrypted

Since secrets are encrypted on write, performing an update on a secret will encrypt that content.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

The command above reads all secrets and then updates them to apply server side encryption. If an error occurs due to a conflicting write, retry the command. For larger clusters, you may wish to subdivide the secrets by namespace or script an update.

Rotating a decryption key

Changing the secret without incurring downtime requires a multi step operation, especially in the presence of a highly available deployment where multiple `kube-apiserver` processes are running.

1. Generate a new key and add it as the second key entry for the current provider on all servers
2. Restart all `kube-apiserver` processes to ensure each server can decrypt using the new key
3. Make the new key the first entry in the `keys` array so that it is used for encryption in the config
4. Restart all `kube-apiserver` processes to ensure each server now encrypts using the new key
5. Run `kubectl get secrets -o json | kubectl replace -f -` to encrypt all existing secrets with the new key
6. Remove the old decryption key from the config after you back up etcd with the new key in use and update all secrets

With a single `kube-apiserver`, step 2 may be skipped.

Decrypting all data

To disable encryption at rest place the `identity` provider as the first entry in the config:

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aescbc:
        keys:
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
```

and restart all `kube-apiserver` processes. Then run the command `kubectl get secrets -o json | kubectl replace -f -` to force all secrets to be decrypted.

Operating etcd clusters for Kubernetes

etcd is a strong, consistent, and highly-available key value store which Kubernetes uses for persistent storage of all of its API objects. This documentation provides specific instruction on operating, upgrading, and rolling back etcd clusters for Kubernetes. For in-depth information on etcd, see etcd documentation.

Prerequisites

- Run etcd as a cluster of odd members.
- etcd is a leader-based distributed system. Ensure that the leader periodically send heartbeats on time to all followers to keep the cluster stable.
- Ensure that no resource starvation occurs.

  Performance and stability of the cluster is sensitive to network and disk IO. Any resource starvation can lead to heartbeat timeout, causing instability of the cluster. An unstable etcd indicates that no leader is elected. Under such circumstances, a cluster cannot make any changes to its current state, which implies no new pods can be scheduled.
- Keeping stable etcd clusters is critical to the stability of Kubernetes clusters. Therefore, run etcd clusters on dedicated machines or isolated environments for guaranteed resource requirements.

Resource requirements

Operating etcd with limited resources is suitable only for testing purposes. For deploying in production, advanced hardware configuration is required. Before deploying etcd in production, see resource requirement reference documentation.

Starting Kubernetes API server

This section covers starting a Kubernetes API server with an etcd cluster in the deployment.

Single-node etcd cluster

Use a single-node etcd cluster only for testing purpose.

1. Run the following:

   ```
   bash ./etcd --client-listen-urls=http://$PRIVATE_IP:2379 --client-advertise-urls=http://$PRIVATE_IP:2379
   ```

2. Start Kubernetes API server with the flag `--etcd-servers=$PRIVATE_IP:2379`.

   Replace `PRIVATE_IP` with your etcd client IP.

Multi-node etcd cluster

For durability and high availability, run etcd as a multi-node cluster in production and back it up periodically. A five-member cluster is recommended in production. For more information, see FAQ Documentation.

Configure an etcd cluster either by static member information or by dynamic discovery. For more information on clustering, see etcd Clustering Documentation.

For an example, consider a five-member etcd cluster running with the following client URLs: `http://$IP1:2379`, `http://$IP2:2379`, `http://$IP3:2379`, `http://$IP4:2379`, and `http://$IP5:2379`. To start a Kubernetes API server:

1. Run the following:

   ```
   ./etcd --client-listen-urls=http://$IP1:2379, http://$IP2:2379, http://$IP3:23
   ```

2. Start Kubernetes API servers with the flag `--etcd-servers=$IP1:2379, $IP2:2379, $IP3:2379, $IP4:2379, $IP5:2379`.

   Replace `IP` with your client IP addresses.

Multi-node etcd cluster with load balancer

To run a load balancing etcd cluster:

1. Set up an etcd cluster.
2. Configure a load balancer in front of the etcd cluster. For example, let the address of the load balancer be `$LB`.
3. Start Kubernetes API Servers with the flag `--etcd-servers=$LB:2379`.

Securing etcd clusters

Access to etcd is equivalent to root permission in the cluster so ideally only the API server should have access to it. Considering the sensitivity of the data, it is recommended to grant permission to only those nodes that require access to etcd clusters.

To secure etcd, either set up firewall rules or use the security features provided by etcd. etcd security features depend on x509 Public Key Infrastructure (PKI). To begin, establish secure communication channels by generating a key and certificate pair. For example, use key pairs `peer.key` and `peer.cert` for securing communication between etcd members, and `client.key` and `client.cert` for securing communication between etcd and its clients. See the example scripts provided by the etcd project to generate key pairs and CA files for client authentication.

Securing communication

To configure etcd with secure peer communication, specify flags `--peer-key-file=peer.key` and `--peer-cert-file=peer.cert`, and use https as URL schema.

Similarly, to configure etcd with secure client communication, specify flags `--key-file=peer.key` and `--cert-file=peer.cert`, and use https as URL schema.

Limiting access of etcd clusters

After configuring secure communication, restrict the access of etcd cluster to only the Kubernetes API server. Use TLS authentication to do so.

For example, consider key pairs `k8sclient.key` and `k8sclient.cert` that are trusted by the CA `etcd.ca`. When etcd is configured with `--client-cert-auth` along with TLS, it verifies the certificates from clients by using system CAs or the CA passed in by `--trusted-ca-file` flag. Specifying flags `--client-cert-auth=true` and `--trust-ca-file=etcd.ca` will restrict the access to clients with the certificate `k8sclient.cert`.

Once etcd is configured correctly, only clients with valid certificates can access it. To give Kubernetes API server the access, configure it with the flags `--etcd-certfile=k8sclient.cert` and `--etcd-keyfile=k8sclient.key`.

Note: etcd authentication is not currently supported by Kubernetes. For more information, see the related issue Support Basic Auth for Etcd v2.

Replacing a failed etcd member

etcd cluster achieves high availability by tolerating minor member failures. However, to improve the overall health of the cluster, replace failed members immediately. When multiple members fail, replace them one by one. Replacing a failed member involves two steps: removing the failed member and adding a new member.

Though etcd keeps unique member IDs internally, it is recommended to use a unique name for each member to avoid human errors. For example, consider a three-member etcd cluster. Let the URLs be, member1=http://10.0.0.1, member2=http://10.0.0.2, and member3=http://10.0.0.3. When member1 fails, replace it with member4=http://10.0.0.4.

1. Get the member ID of the failed member1:

   ```
   etcdctl --endpoints=http://10.0.0.2,http://10.0.0.3 member list
   ```

   The following message is displayed:

   ```
   8211f1d0f64f3269, started, member1, http://10.0.0.1:12380, http://10.0.0.1:23
   91bc3c398fb3c146, started, member2, http://10.0.0.1:2380, http://10.0.0.2:237
   fd422379fda50e48, started, member3, http://10.0.0.1:2380, http://10.0.0.3:237
   ```

2. Remove the failed member:

   ```
   etcdctl member remove 8211f1d0f64f3269
   ```

   The following message is displayed:

   ```
   Removed member 8211f1d0f64f3269 from cluster
   ```

3. Add the new member:

   ```
   ./etcdctl member add member4 --peer-urls=http://10.0.0.4:2380
   ```

   The following message is displayed:

   ```
   Member 2be1eb8f84b7f63e added to cluster ef37ad9dc622a7c4
   ```

4. Start the newly added member on a machine with the IP `10.0.0.4`:

   ```
   bash export ETCD_NAME="member4" export
   ETCD_INITIAL_CLUSTER="member2=http://10.0.0.2:2380,member3=http://10.0.0.3:2380,me
   export ETCD_INITIAL_CLUSTER_STATE=existing etcd [flags]
   ```

5. Do either of the following:

   1. Update its `--etcd-servers` flag to make Kubernetes aware of the configuration changes, then restart the Kubernetes API server.

   2. Update the load balancer configuration if a load balancer is used in the deployment.

For more information on cluster reconfiguration, see etcd Reconfiguration Documentation.

Backing up an etcd cluster

All Kubernetes objects are stored on etcd. Periodically backing up the etcd cluster data is important to recover Kubernetes clusters under disaster scenarios, such as losing all master nodes. The snapshot file contains all the Kubernetes states and critical information. In order to keep the sensitive Kubernetes data safe, encrypt the snapshot files.

Backing up an etcd cluster can be accomplished in two ways: etcd built-in snapshot and volume snapshot.

Built-in snapshot

etcd supports built-in snapshot, so backing up an etcd cluster is easy. A snapshot may either be taken from a live member with the `etcdctl snapshot save` command or by copying the `member/snap/db` file from an etcd data directory that is not currently used by an etcd process. `datadir` is located at `$DATA_DIR/member/snap/db`. Taking the snapshot will normally not affect the performance of the member.

Below is an example for taking a snapshot of the keyspace served by `$ENDPOINT` to the file `snapshotdb`:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshotdb
# exit 0

# verify the snapshot
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshotdb
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| fe01cf57 |       10 |          7 | 2.1 MB     |
+----------+----------+------------+------------+
```

Volume snapshot

If etcd is running on a storage volume that supports backup, such as Amazon Elastic Block Store, back up etcd data by taking a snapshot of the storage volume.

Scaling up etcd clusters

Scaling up etcd clusters increases availability by trading off performance. Scaling does not increase cluster performance nor capability. A general rule is not to scale up or down etcd clusters. Do not configure any auto scaling groups for etcd clusters. It is highly recommended to always run a static five-member etcd cluster for production Kubernetes clusters at any officially supported scale.

A reasonable scaling is to upgrade a three-member cluster to a five-member one, when more reliability is desired. See etcd Reconfiguration Documentation for information on how to add members into an existing cluster.

Restoring an etcd cluster

etcd supports restoring from snapshots that are taken from an etcd process of the major.minor version. Restoring a version from a different patch version of etcd also is supported. A restore operation is employed to recover the data of a failed cluster.

Before starting the restore operation, a snapshot file must be present. It can either be a snapshot file from a previous backup operation, or from a remaining data directory. `datadir` is located at `$DATA_DIR/member/snap/db`. For more information and examples on restoring a cluster from a snapshot file, see etcd disaster recovery documentation.

If the access URLs of the restored cluster is changed from the previous cluster, the Kubernetes API server must be reconfigured accordingly. In this case, restart Kubernetes API server with the flag `--etcd-servers=$NEW_ETCD_CLUSTER` instead of the flag `--etcd-servers=$OLD_ETCD_CLUSTER`. Replace `$NEW_ETCD_CLUSTER` and `$OLD_ETCD_CLUSTER` with the respective IP addresses. If a load balancer is used in front of an etcd cluster, you might need to update the load balancer instead.

If the majority of etcd members have permanently failed, the etcd cluster is considered failed. In this scenario, Kubernetes cannot make any changes to its current state. Although the scheduled pods might continue to run, no new pods can be scheduled. In such cases, recover the etcd cluster and potentially reconfigure Kubernetes API server to fix the issue.

Upgrading and rolling back etcd clusters

Important assumptions

The upgrade procedure described in this document assumes that either:

1. The etcd cluster has only a single node.
2. The etcd cluster has multiple nodes.

In this case, the upgrade procedure requires shutting down the etcd cluster. During the time the etcd cluster is shut down, the Kubernetes API Server will be read only.

Warning: Deviations from the assumptions are untested by continuous integration, and deviations might create undesirable consequences. Additional information about operating an etcd cluster is available from the etcd maintainers.

Background

As of Kubernetes version 1.5.1, we are still using etcd from the 2.2.1 release with the v2 API. Also, we have no pre-existing process for updating etcd, as we have never updated etcd by either minor or major version.

Note that we need to migrate both the etcd versions that we are using (from 2.2.1 to at least 3.0.x) as well as the version of the etcd API that Kubernetes talks to. The etcd 3.0.x binaries support both the v2 and v3 API.

This document describes how to do this migration. If you want to skip the background and cut right to the procedure, see Upgrade Procedure.

etcd upgrade requirements

There are requirements on how an etcd cluster upgrade can be performed. The primary considerations are: - Upgrade between one minor release at a time - Rollback supported through additional tooling

One minor release at a time

Upgrade only one minor release at a time. For example, we cannot upgrade directly from 2.1.x to 2.3.x. Within patch releases it is possible to upgrade and downgrade between arbitrary versions. Starting a cluster for any intermediate minor release, waiting until the cluster is healthy, and then shutting down the cluster down will perform the migration. For example, to upgrade from version 2.1.x to 2.3.y, it is enough to start etcd in 2.2.z version, wait until it is healthy, stop it, and then start the 2.3.y version.

Rollback via additional tooling

Versions 3.0+ of etcd do not support general rollback. That is, after migrating from M.N to M.N+1, there is no way to go back to M.N. The etcd team has provided a custom rollback tool but the rollback tool has these limitations:

- This custom rollback tool is not part of the etcd repo and does not receive the same testing as the rest of etcd. We are testing it in a couple of end-to-end tests. There is only community support here.
- The rollback can be done only from the 3.0.x version (that is using the v3 API) to the 2.2.1 version (that is using the v2 API).
- The tool only works if the data is stored in `application/json` format.
- Rollback doesn't preserve resource versions of objects stored in etcd.

Warning: If the data is not kept in `application/json` format (see Upgrade Procedure), you will lose the option to roll back to etcd 2.2.

The last bullet means that any component or user that has some logic depending on resource versions may require restart after etcd rollback. This includes that all clients using the watch API, which depends on resource versions. Since both the kubelet and kube-proxy use the watch API, a rollback might require restarting all Kubernetes components on all nodes.

Note: At the time of writing, both Kubelet and KubeProxy are using "resource version" only for watching (i.e. are not using resource versions for anything else). And both are using reflector and/or informer frameworks for watching (i.e. they don't send watch requests themselves). Both those frameworks if they can't renew watch, they will start from "current version" by doing "list + watch from the resource version returned by list". That means that if the apiserver will be down for the period of rollback, all of node components should basically restart their watches and start from "now" when apiserver is back. And it will be back with new resource version. That would mean that restarting node components is not needed. But the assumptions here may not hold forever.

Design

This section describes how we are going to do the migration, given the etcd upgrade requirements.

Note that because the code changes in Kubernetes code needed to support the etcd v3 API are local and straightforward, we do not focus on them at all. We focus only on the upgrade/rollback here.

New etcd Docker image

We decided to completely change the content of the etcd image and the way it works. So far, the Docker image for etcd in version X has contained only the etcd and etcdctl binaries.

Going forward, the Docker image for etcd in version X will contain multiple versions of etcd. For example, the 3.0.17 image will contain the 2.2.1, 2.3.7, and 3.0.17 binaries of etcd and etcdctl. This will allow running etcd in multiple different versions using the same Docker image.

Additionally, the image will contain a custom script, written by the Kubernetes team, for doing migration between versions. The image will also contain the rollback tool provided by the etcd team.

Migration script

The migration script that will be part of the etcd Docker image is a bash script that works as follows:

1. Detect which version of etcd we were previously running. For that purpose, we have added a dedicated file, `version.txt`, that holds that information and is stored in the etcd-data-specific directory, next to the etcd data. If the file doesn't exist, we default it to version 2.2.1.
2. If we are in version 2.2.1 and are supposed to upgrade, backup data.
3. Based on the detected previous etcd version and the desired one (communicated via environment variable), do the upgrade steps as needed. This means that for every minor etcd release greater than the detected one and less than or equal to the desired one:
    1. Start etcd in that version.
    2. Wait until it is healthy. Healthy means that you can write some data to it.
    3. Stop this etcd. Note that this etcd will not listen on the default etcd port. It is hard coded to listen on ports that the API server is not configured to connect to, which means that API server won't be able to connect to it. Assuming no other client goes out of its way to try to connect and write to this obscure port, no new data will be written during this period.
4. If the desired API version is v3 and the detected version is v2, do the offline migration from the v2 to v3 data format. For that we use two tools:
    1. ./etcdctl migrate: This is the official tool for migration provided by the etcd team.

   2. A custom script that is attaching TTLs to events in the etcd. Note that etcdctl migrate doesn't support TTLs.

5. After every successful step, update contents of the version file. This will protect us from the situation where something crashes in the meantime ,and the version file gets completely unsynchronized with the real data. Note that it is safe if the script crashes after the step is done and before the file is updated. This will only result in redoing one step in the next try.

All the previous steps are for the case where the detected version is less than or equal to the desired version. In the opposite case, that is for a rollback, the script works as follows:

1. Verify that the detected version is 3.0.x with the v3 API, and the desired version is 2.2.1 with the v2 API. We don't support any other rollback.
2. If so, we run the custom tool provided by etcd team to do the offline rollback. This tool reads the v3 formatted data and writes it back to disk in v2 format.
3. Finally update the contents of the version file.

Upgrade procedure

Simply modify the command line in the etcd manifest to:

1. Run the migration script. If the previously run version is already in the desired version, this will be no-op.
2. Start etcd in the desired version.

Starting in Kubernetes version 1.6, this has been done in the manifests for new Google Compute Engine clusters. You should also specify these environment variables. In particular,you must keep `STORAGE_MEDIA_TYPE` set to `application/json` if you wish to preserve the option to roll back.

```
TARGET_STORAGE=etcd3
ETCD_IMAGE=3.0.17
TARGET_VERSION=3.0.17
STORAGE_MEDIA_TYPE=application/json
```

To roll back, use these:

```
TARGET_STORAGE=etcd2
ETCD_IMAGE=3.0.17
TARGET_VERSION=2.2.1
STORAGE_MEDIA_TYPE=application/json
```

Notes for etcd Version 2.2.1

Default configuration

The default setup scripts use kubelet's file-based static pods feature to run etcd in a pod. This manifest should only be run on master VMs. The default location that kubelet scans for manifests is `/etc/kubernetes/manifests/` .

Kubernetes's usage of etcd

By default, Kubernetes objects are stored under the `/registry` key in etcd. This path can be prefixed by using the kube-apiserver flag `--etcd-prefix="/foo"` .

`etcd` is the only place that Kubernetes keeps state.

Troubleshooting

To test whether `etcd` is running correctly, you can try writing a value to a test key. On your master VM (or somewhere with firewalls configured such that you can talk to your cluster's etcd), try:

```
curl -X PUT "http://${host}:${port}/v2/keys/_test"
```

Static Pods

If you are running clustered Kubernetes and are using static pods to run a pod on every node, you should probably be using a DaemonSet!

Static pods are managed directly by kubelet daemon on a specific node, without API server observing it. It does not have associated any replication controller, kubelet daemon itself watches it and restarts it when it crashes. There is no health check though. Static pods are always bound to one kubelet daemon and always run on the same node with it.

Kubelet automatically creates so-called *mirror pod* on Kubernetes API server for each static pod, so the pods are visible there, but they cannot be controlled from the API server.

Static pod creation

Static pod can be created in two ways: either by using configuration file(s) or by HTTP.

Configuration files

The configuration files are just standard pod definition in json or yaml format in specific directory. Use `kubelet --pod-manifest-path=<the directory>` to start kubelet daemon, which periodically scans the directory and creates/deletes static pods as yaml/json files appear/disappear there. Note that kubelet will ignore files starting with dots when scanning the specified directory.

For example, this is how to start a simple web server as a static pod:

1. Choose a node where we want to run the static pod. In this example, it's `my-node1`.

   ```
   [joe@host ~] $ ssh my-node1
   ```

2. Choose a directory, say `/etc/kubelet.d` and place a web server pod definition there, e.g. `/etc/kubelet.d/static-web.yaml`:

```
[root@my-node1 ~] $ mkdir /etc/kubelet.d/ [root@my-node1 ~] $ cat <<EOF
>/etc/kubelet.d/static-web.yaml apiVersion: v1 kind: Pod metadata: name:
static-web labels: role: myrole spec: containers: - name: web image: nginx
ports: - name: web containerPort: 80 protocol: TCP EOF
```

3. Configure your kubelet daemon on the node to use this directory by running it with `--pod-manifest-path=/etc/kubelet.d/` argument. On Fedora edit `/etc/kubernetes/kubelet` to include this line:

   ```
   KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --pod-
   manifest-path=/etc/kubelet.d/"
   ```

   Instructions for other distributions or Kubernetes installations may vary.

4. Restart kubelet. On Fedora, this is:

   ```
   [root@my-node1 ~] $ systemctl restart kubelet
   ```

Pods created via HTTP

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a json/yaml file with a pod definition. It works the same as `--pod-manifest-path=<directory>`, i.e. it's reloaded every now and then and changes are applied to running static pods (see below).

Behavior of static pods

When kubelet starts, it automatically starts all pods defined in directory specified in `--pod-manifest-path=` or `--manifest-url=` arguments, i.e. our static-web. (It may take some time to pull nginx image, be patient…):

```
[joe@my-node1 ~] $ docker ps
CONTAINER ID IMAGE         COMMAND  CREATED        STATUS         PORTS     NAMES
f6d05272b57e nginx:latest  "nginx"  8 minutes ago  Up 8 minutes             k8s_we
```

If we look at our Kubernetes API server (running on host `my-master`), we see that a new mirror-pod was created there too:

```
[joe@host ~] $ ssh my-master
[joe@my-master ~] $ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
static-web-my-node1         1/1       Running   0          2m
```

Labels from the static pod are propagated into the mirror-pod and can be used as usual for filtering.

Notice we cannot delete the pod with the API server (e.g. via `kubectl` command), kubelet simply won't remove it.

```
[joe@my-master ~] $ kubectl delete pod static-web-my-node1
pod "static-web-my-node1" deleted
[joe@my-master ~] $ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
static-web-my-node1         1/1       Running   0          12s
```

Back to our `my-node1` host, we can try to stop the container manually and see, that kubelet automatically restarts it in a while:

```
[joe@host ~] $ ssh my-node1
[joe@my-node1 ~] $ docker stop f6d05272b57e
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
CONTAINER ID        IMAGE         COMMAND                CREATED       ...
5b920cbaf8b1        nginx:latest  "nginx -g 'daemon of   2 seconds ago ...
```

Dynamic addition and removal of static pods

Running kubelet periodically scans the configured directory (`/etc/kubelet.d` in our example) for changes and adds/removes pods as files appear/disappear in this directory.

```
[joe@my-node1 ~] $ mv /etc/kubelet.d/static-web.yaml /tmp
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
// no nginx container is running
[joe@my-node1 ~] $ mv /tmp/static-web.yaml  /etc/kubelet.d/
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
CONTAINER ID        IMAGE         COMMAND                CREATED                 ...
e7a62e3427f1        nginx:latest  "nginx -g 'daemon of   27 seconds ago
```

Cluster Management

This document describes several topics related to the lifecycle of a cluster: creating a new cluster, upgrading your cluster's master and worker nodes, performing node maintenance (e.g. kernel upgrades), and upgrading the Kubernetes API version of a running cluster.

Creating and configuring a Cluster

To install Kubernetes on a set of machines, consult one of the existing Getting Started guides depending on your environment.

Upgrading a cluster

The current state of cluster upgrades is provider dependent, and some releases may require special care when upgrading. It is recommended that administrators consult both the release notes, as well as the version specific upgrade notes prior to upgrading their clusters.

- Upgrading to 1.6

Upgrading Google Compute Engine clusters

Google Compute Engine Open Source (GCE-OSS) support master upgrades by deleting and recreating the master, while maintaining the same Persistent Disk (PD) to ensure that data is retained across the upgrade.

Node upgrades for GCE use a Managed Instance Group, each node is sequentially destroyed and then recreated with new software. Any Pods that are running on that node need to be controlled by a Replication Controller, or manually re-created after the roll out.

Upgrades on open source Google Compute Engine (GCE) clusters are controlled by the `cluster/gce/upgrade.sh` script.

Get its usage by running `cluster/gce/upgrade.sh -h`.

For example, to upgrade just your master to a specific version (v1.0.2):

```
cluster/gce/upgrade.sh -M v1.0.2
```

Alternatively, to upgrade your entire cluster to the latest stable release:

```
cluster/gce/upgrade.sh release/stable
```

Upgrading Google Container Engine (GKE) clusters

Google Container Engine automatically updates master components (e.g. `kube-apiserver`, `kube-scheduler`) to the latest version. It also handles upgrading the operating system and other components that the master runs on.

The node upgrade process is user-initiated and is described in the GKE documentation.

Upgrading clusters on other platforms

Different providers, and tools, will manage upgrades differently. It is recommended that you consult their main documentation regarding upgrades.

- kops
- kubespray

- CoreOS Tectonic
- ...

Resizing a cluster

If your cluster runs short on resources you can easily add more machines to it if your cluster is running in Node self-registration mode. If you're using GCE or GKE it's done by resizing Instance Group managing your Nodes. It can be accomplished by modifying number of instances on `Compute > Compute Engine > Instance groups > your group > Edit group` Google Cloud Console page or using gcloud CLI:

```
gcloud compute instance-groups managed resize kubernetes-minion-group --size=42 --
```

Instance Group will take care of putting appropriate image on new machines and start them, while Kubelet will register its Node with API server to make it available for scheduling. If you scale the instance group down, system will randomly choose Nodes to kill.

In other environments you may need to configure the machine yourself and tell the Kubelet on which machine API server is running.

Cluster autoscaling

If you are using GCE or GKE, you can configure your cluster so that it is automatically rescaled based on pod needs.

As described in Compute Resource, users can reserve how much CPU and memory is allocated to pods. This information is used by the Kubernetes scheduler to find a place to run the pod. If there is no node that has enough free capacity (or doesn't match other pod requirements) then the pod has to wait until some pods are terminated or a new node is added.

Cluster autoscaler looks for the pods that cannot be scheduled and checks if adding a new node, similar to the other in the cluster, would help. If yes, then it resizes the cluster to accommodate the waiting pods.

Cluster autoscaler also scales down the cluster if it notices that some node is not needed anymore for an extended period of time (10min but it may change in the future).

Cluster autoscaler is configured per instance group (GCE) or node pool (GKE).

If you are using GCE then you can either enable it while creating a cluster with kube-up.sh script. To configure cluster autoscaler you have to set three environment variables:

- `KUBE_ENABLE_CLUSTER_AUTOSCALER` - it enables cluster autoscaler if set to true.
- `KUBE_AUTOSCALER_MIN_NODES` - minimum number of nodes in the cluster.
- `KUBE_AUTOSCALER_MAX_NODES` - maximum number of nodes in the cluster.

Example:

```
KUBE_ENABLE_CLUSTER_AUTOSCALER=true KUBE_AUTOSCALER_MIN_NODES=3 KUBE_AUTOSCALER_MA
```

On GKE you configure cluster autoscaler either on cluster creation or update or when creating a particular node pool (which you want to be autoscaled) by passing flags `--enable-autoscaling` `--min-nodes` and `--max-nodes` to the corresponding `gcloud` commands.

Examples:

```
gcloud container clusters create mytestcluster --zone=us-central1-b --enable-autos
```

```
gcloud container clusters update mytestcluster --enable-autoscaling --min-nodes=1
```

Cluster autoscaler expects that nodes have not been manually modified (e.g. by adding labels via kubectl) as those properties would not be propagated to the new nodes within the same instance group.

Maintenance on a Node

If you need to reboot a node (such as for a kernel upgrade, libc upgrade, hardware repair, etc.), and the downtime is brief, then when the Kubelet restarts, it will attempt to restart the pods scheduled to it. If the reboot takes longer (the default time is 5 minutes, controlled by `--pod-eviction-timeout` on the controller-manager), then the node controller will terminate the pods that are bound to the unavailable node. If there is a corresponding replica set (or replication controller), then a new copy of the pod will be started on a different node. So, in the case where all pods are replicated, upgrades can be done without special coordination, assuming that not all nodes will go down at the same time.

If you want more control over the upgrading process, you may use the following workflow:

Use `kubectl drain` to gracefully terminate all pods on the node while marking the node as unschedulable:

```
kubectl drain $NODENAME
```

This keeps new pods from landing on the node while you are trying to get them off.

For pods with a replica set, the pod will be replaced by a new pod which will be scheduled to a new node. Additionally, if the pod is part of a service, then clients will automatically be redirected to the new pod.

For pods with no replica set, you need to bring up a new copy of the pod, and assuming it is not part of a service, redirect clients to it.

Perform maintenance work on the node.

Make the node schedulable again:

```
kubectl uncordon $NODENAME
```

If you deleted the node's VM instance and created a new one, then a new schedulable node resource will be created automatically (if you're using a cloud provider that supports node discovery; currently this is only Google Compute Engine, not including CoreOS on Google Compute Engine using kube-register). See Node for more details.

Advanced Topics

Upgrading to a different API version

When a new API version is released, you may need to upgrade a cluster to support the new API version (e.g. switching from ‘v1’ to ‘v2’ when ‘v2’ is launched).

This is an infrequent event, but it requires careful management. There is a sequence of steps to upgrade to a new API version.

1. Turn on the new API version.
2. Upgrade the cluster’s storage to use the new version.
3. Upgrade all config files. Identify users of the old API version endpoints.
4. Update existing objects in the storage to new version by running `cluster/update-storage-objects.sh`.
5. Turn off the old API version.

Turn on or off an API version for your cluster

Specific API versions can be turned on or off by passing `--runtime-config=api/<version>` flag while bringing up the API server. For example: to turn off v1 API, pass `--runtime-config=api/v1=false`. runtime-config also supports 2 special keys: api/all and api/legacy to control all and legacy APIs respectively. For example, for turning off all API versions except v1, pass `--runtime-config=api/all=false,api/v1=true`. For the purposes of these flags, *legacy* APIs are those APIs which have been explicitly deprecated (e.g. `v1beta3`).

Switching your cluster’s storage API version

The objects that are stored to disk for a cluster’s internal representation of the Kubernetes resources active in the cluster are written using a particular version of the API. When the supported API changes, these objects may need to be rewritten in the newer API. Failure to do this will eventually result in resources that are no longer decodable or usable by the Kubernetes API server.

`KUBE_API_VERSIONS` environment variable for the `kube-apiserver` binary which controls the API versions that are supported in the cluster. The first version in the list is used as the cluster’s storage version. Hence, to set a specific version as the storage version, bring it to the front of list of versions in the value of `KUBE_API_VERSIONS`. You need to restart the `kube-apiserver` binary for changes to this variable to take effect.

Switching your config files to a new API version

You can use `kubectl convert` command to convert config files between different API versions.

```
kubectl convert -f pod.yaml --output-version v1
```

For more options, please refer to the usage of kubectl convert command.

Cluster Management Guide for Version 1.6

This document outlines the potentially disruptive changes that exist in the 1.6 release cycle. Operators, administrators, and developers should take note of the changes below in order to maintain continuity across their upgrade process.

Cluster defaults set to etcd 3

In the 1.6 release cycle, the default backend storage layer has been upgraded to fully leverage etcd 3 capabilities by default. For new clusters, there is nothing an operator will need to do, it should "just work". However, if you are upgrading from a 1.5 cluster, care should be taken to ensure continuity.

It is possible to maintain v2 compatibility mode while running etcd 3 for an interim period of time. To do this, you will simply need to update an argument passed to your apiserver during startup:

```
$ kube-apiserver --storage-backend='etcd2' $(EXISTING_ARGS)
```

However, for long-term maintenance of the cluster, we recommend that the operator plan an outage window in order to perform a v2->v3 data upgrade.

Upgrading kubeadm clusters from 1.6 to 1.7

This guide is for upgrading kubeadm clusters from version 1.6.x to 1.7.x. Upgrades are not supported for clusters lower than 1.6, which is when kubeadm became Beta.

WARNING: These instructions will **overwrite** all of the resources managed by kubeadm (static pod manifest files, service accounts and RBAC rules in the `kube-system` namespace, etc.), so any customizations you may have made to these resources after cluster setup will need to be reapplied after the upgrade. The upgrade will not disturb other static pod manifest files or objects outside the `kube-system` namespace.

Before you begin

You need to have a Kubernetes cluster running version 1.6.x.

On the master

1. Upgrade system packages.

   Upgrade your OS packages for kubectl, kubeadm, kubelet, and kubernetes-cni.

   a. On Debian, this can be accomplished with:

   ```
   sudo apt-get update
   sudo apt-get upgrade
   ```

   b. On CentOS/Fedora, you would instead run:

```
sudo yum update
```

2. Restart kubelet.

```
systemctl restart kubelet
```

3. Delete the `kube-proxy` DaemonSet.

   Although most components are automatically upgraded by the next step, `kube-proxy` currently needs to be manually deleted so it can be recreated at the correct version:

```
sudo KUBECONFIG=/etc/kubernetes/admin.conf kubectl delete daemonset kube-proxy
```

4. Perform kubeadm upgrade.

   WARNING: All parameters you passed to the first `kubeadm init` when you bootstrapped your cluster **MUST** be specified here in the upgrade-`kubeadm init`-command. This is a limitation we plan to address in v1.8.

```
sudo kubeadm init --skip-preflight-checks --kubernetes-version <DESIRED_VERSIO
```

   For instance, if you want to upgrade to `1.7.0`, you would run:

```
sudo kubeadm init --skip-preflight-checks --kubernetes-version v1.7.0
```

5. Upgrade CNI provider.

   Your CNI provider might have its own upgrade instructions to follow now. Check the addons page to find your CNI provider and see if there are additional upgrade steps necessary.

On each node

1. Upgrade system packages.

   Upgrade your OS packages for kubectl, kubeadm, kubelet, and kubernetes-cni.

   a. On Debian, this can be accomplished with:

   ```
   sudo apt-get update
   sudo apt-get upgrade
   ```

   b. On CentOS/Fedora, you would instead run:

   ```
   sudo yum update
   ```

2. Restart kubelet.

   ```
   systemctl restart kubelet
   ```

Upgrading kubeadm clusters from 1.7 to 1.8

This guide is for upgrading `kubeadm` clusters from version 1.7.x to 1.8.x, as well as 1.7.x to 1.7.y and 1.8.x to 1.8.y where `y > x`. See also upgrading kubeadm clusters from 1.6 to 1.7 if you're on a 1.6 cluster currently.

Before you begin

Before proceeding:

- You need to have a functional `kubeadm` Kubernetes cluster running version 1.7.0 or higher in order to use the process described here.
- Make sure you read the release notes carefully.
- As `kubeadm upgrade` does not upgrade etcd make sure to back it up. You can, for example, use `etcdctl backup` to take care of this.
- Note that `kubeadm upgrade` will not touch any of your workloads, only Kubernetes-internal components. As a best-practice you should back up what's important to you. For example, any app-level state, such as a database an app might depend on (like MySQL or MongoDB) must be backed up beforehand.

Also, note that only one minor version upgrade is supported. That is, you can only upgrade from, say 1.7 to 1.8, not from 1.7 to 1.9.

Upgrading your control plane

You have to carry out the following steps by executing these commands on your master node:

1. Install the most recent version of `kubeadm` using `curl` like so:

```
$ export VERSION=$(curl -sSL https://dl.k8s.io/release/stable.txt) # or manually s
$ export ARCH=amd64 # or: arm, arm64, ppc64le, s390x
$ curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > /usr/
$ chmod a+rx /usr/bin/kubeadm
```

Verify that this download of kubeadm works, and has the expected version:

```
$ kubeadm version
```

1. If this the first time you use `kubeadm upgrade`, in order to preserve the configuration for future upgrades, do:

Note that for below you will need to recall what CLI args you passed to `kubeadm init` the first time.

If you used flags, do:

```
$ kubeadm config upload from-flags [flags]
```

Where `flags` can be empty.

If you used a config file, do:

```
$ kubeadm config upload from-file --config [config]
```

Where the `config` is mandatory.

1. On the master node, run the following:

```
$ kubeadm upgrade plan
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist
```

```
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-syste
[upgrade] Fetching available versions to upgrade to:
[upgrade/versions] Cluster version: v1.7.1
[upgrade/versions] kubeadm version: v1.8.0
[upgrade/versions] Latest stable version: v1.8.0
[upgrade/versions] Latest version in the v1.7 series: v1.7.6

Components that must be upgraded manually after you've upgraded the control plane
COMPONENT   CURRENT     AVAILABLE
Kubelet     1 x v1.7.1  v1.7.6

Upgrade to the latest version in the v1.7 series:

COMPONENT            CURRENT   AVAILABLE
API Server           v1.7.1    v1.7.6
Controller Manager   v1.7.1    v1.7.6
Scheduler            v1.7.1    v1.7.6
Kube Proxy           v1.7.1    v1.7.6
Kube DNS             1.14.4    1.14.4

You can now apply the upgrade by executing the following command:

        kubeadm upgrade apply v1.7.6

_____________________________________________________________________

Components that must be upgraded manually after you've upgraded the control plane
COMPONENT   CURRENT     AVAILABLE
Kubelet     1 x v1.7.1  v1.8.0

Upgrade to the latest experimental version:

COMPONENT            CURRENT   AVAILABLE
API Server           v1.7.1    v1.8.0
Controller Manager   v1.7.1    v1.8.0
Scheduler            v1.7.1    v1.8.0
Kube Proxy           v1.7.1    v1.8.0
Kube DNS             1.14.4    1.14.4

You can now apply the upgrade by executing the following command:

        kubeadm upgrade apply v1.8.0

Note: Before you do can perform this upgrade, you have to update kubeadm to v1.8.0

_____________________________________________________________________
```

The `kubeadm upgrade plan` checks that your cluster is in an upgradeable state and fetches the versions available to upgrade to in an user-friendly way.

1. Pick a version to upgrade to and run, for example, `kubeadm upgrade apply` as follows:

```
$ kubeadm upgrade apply v1.8.0
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-syste
[upgrade/version] You have chosen to upgrade to version "v1.8.0"
[upgrade/versions] Cluster version: v1.7.1
[upgrade/versions] kubeadm version: v1.8.0
[upgrade/prepull] Will prepull images for components [kube-apiserver kube-controll
[upgrade/prepull] Prepulling image for component kube-scheduler.
[upgrade/prepull] Prepulling image for component kube-apiserver.
[upgrade/prepull] Prepulling image for component kube-controller-manager.
[apiclient] Found 0 Pods for label selector k8s-app=upgrade-prepull-kube-scheduler
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-scheduler
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-apiserver
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-controlle
[upgrade/prepull] Prepulled image for component kube-apiserver.
[upgrade/prepull] Prepulled image for component kube-controller-manager.
[upgrade/prepull] Prepulled image for component kube-scheduler.
[upgrade/prepull] Successfully prepulled the images for all the control plane comp
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version "v1.8.0"
[upgrade/staticpods] Writing upgraded Static Pod manifests to "/etc/kubernetes/tmp
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kub
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to 
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kub
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-ap
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-apiserver
[upgrade/staticpods] Component "kube-apiserver" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-co
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-controller-manager
[upgrade/staticpods] Component "kube-controller-manager" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-sc
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-scheduler
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automat
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.8.0". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upg
```

`kubeadm upgrade apply` does the following:

- It checks that your cluster is in an upgradeable state, that is:
  - The API Server is reachable,
  - All nodes are in the `Ready` state, and
  - The control plane is healthy
- It enforces the version skew policies.
- It makes sure the control plane images are available or available to pull to the machine.
- It upgrades the control plane components or rollbacks if any of them fails to come up.
- It applies the new `kube-dns` and `kube-proxy` manifests and enforces that all necessary RBAC rules are created.

1. Manually upgrade your Software Defined Network (SDN).

   Your Container Network Interface (CNI) provider might have its own upgrade instructions to follow now. Check the addons page to find your CNI provider and see if there are additional upgrade steps necessary.

Upgrading your master and node packages

For each host (referred to as `$HOST` below) in your cluster, upgrade `kubelet` by executing the following commands:

1. Prepare the host for maintenance, marking it unschedulable and evicting the workload:

```
$ kubectl drain $HOST --ignore-daemonsets
```

When running this command against the master host, this error is expected and can be safely ignored (since there are static pods running on the master):

```
node "master" already cordoned
error: pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or St
```

1. Upgrade the Kubernetes package versions on the `$HOST` node by using a Linux distribution-specific package manager:

If the host is running a Debian-based distro such as Ubuntu, run:

```
$ apt-get update
$ apt-get upgrade
```

If the host is running CentOS or the like, run:

```
$ yum update
```

Now the new version of the `kubelet` should be running on the host. Verify this using the following command on `$HOST` :

```
$ systemctl status kubelet
```

1. Since certificate rotation is enabled by default, you may need to manually approve the new kubelet’s CertificateSigningRequest before it can rejoin the cluster:

```
$ kubectl get csr | grep -v Approved
NAME                                                   AGE       REQUESTOR
node-csr-cz132tarZb_XYKnvXf0Q0o4spGUXzJhN2p4_ld7k1iM   2h        system:bootstrap:
```

If you see any CSRs listed that aren’t already approved, you can manually approve them using kubectl:

```
$ kubectl certificate approve node-csr-cz132tarZb_XYKnvXf0Q0o4spGUXzJhN2p4_ld7k1iM
certificatesigningrequest "node-csr-cz132tarZb_XYKnvXf0Q0o4spGUXzJhN2p4_ld7k1iM" a
```

1. Bring the host back online by marking it schedulable:

```
$ kubectl uncordon $HOST
```

1. After upgrading `kubelet` on each host in your cluster, verify that all nodes are available again by executing the following (from anywhere, for example, from outside the cluster):

```
$ kubectl get nodes
```

If the `STATUS` column of the above command shows `Ready` for all of your hosts, you are done.

Recovering from a bad state

If `kubeadm upgrade` somehow fails and fails to roll back, due to an unexpected shutdown during execution for instance, you may run `kubeadm upgrade` again as it is idempotent and should eventually make sure the actual state is the desired state you are declaring.

You can use `kubeadm upgrade` to change a running cluster with `x.x.x --> x.x.x` with `--force`, which can be used to recover from a bad state.

Share a Cluster with Namespaces

This page shows how to view, work in, and delete namespaces. The page also shows how to use Kubernetes namespaces to subdivide your cluster.

Before you begin

- Have an existing Kubernetes cluster.
- Have a basic understanding of Kubernetes *Pods*, *Services*, and *Deployments*.

Viewing namespaces

1. List the current namespaces in a cluster using:

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    11d
kube-system   Active    11d
```

Kubernetes starts with two initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system

You can also get the summary of a specific namespace using:

```
$ kubectl get namespaces <name>
```

Or you can get detailed information with:

```
$ kubectl describe namespaces <name>
Name:           default
Labels:         <none>
Annotations:    <none>
Status:         Active

No resource quota.

Resource Limits
 Type       Resource    Min Max Default
 ----               --------    --- --- ---
 Container          cpu         -   -   100m
```

Note that these details show both resource quota (if present) as well as resource limit ranges.

Resource quota tracks aggregate usage of resources in the *Namespace* and allows cluster operators to define *Hard* resource usage limits that a *Namespace* may consume.

A limit range defines min/max constraints on the amount of resources a single entity can consume in a *Namespace*.

See Admission control: Limit Range

A namespace can be in one of two phases:

- `Active` the namespace is in use
- `Terminating` the namespace is being deleted, and can not be used for new objects

See the design doc for more details.

Creating a new namespace

1. Create a new YAML file called `my-namespace.yaml` with the contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

Then run:

```
$ kubectl create -f ./my-namespace.yaml
```

Note that the name of your namespace must be a DNS compatible label.

There's an optional field `finalizers`, which allows observables to purge resources whenever the namespace is deleted. Keep in mind that if you specify a nonexistent finalizer, the namespace will be created but will get stuck in the `Terminating` state if the user tries to delete it.

More information on `finalizers` can be found in the namespace design doc.

Deleting a namespace

1. Delete a namespace with

```
$ kubectl delete namespaces <insert-some-namespace-name>
```

WARNING, this deletes *everything* under the namespace!

This delete is asynchronous, so for a time you will see the namespace in the `Terminating` state.

Subdividing your cluster using Kubernetes namespaces

1. Understand the default namespace

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

Assuming you have a fresh cluster, you can introspect the available namespace's by doing the following:

```
$ kubectl get namespaces
NAME      STATUS    AGE
default   Active    13m
```

1. Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

In a scenario where an organization is using a shared Kubernetes cluster for development and production use cases:

The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: development and production.

Let’s create two new namespaces to hold our work.

Use the file `namespace-dev.json` which describes a development namespace:

namespace-dev.json

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "development",
    "labels": {
      "name": "development"
    }
  }
}
```

Create the development namespace using kubectl.

```
$ kubectl create -f docs/admin/namespaces/namespace-dev.json
```

And then let's create the production namespace using kubectl.

```
$ kubectl create -f docs/admin/namespaces/namespace-prod.json
```

To be sure things are right, list all of the namespaces in our cluster.

```
$ kubectl get namespaces --show-labels
NAME          STATUS    AGE       LABELS
default       Active    32m       <none>
development   Active    29s       name=development
production    Active    23s       name=production
```

1. Create pods in each namespace

A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster.

Users interacting with one namespace do not see the content in another namespace.

To demonstrate this, let's spin up a simple Deployment and Pods in the development namespace.

We first check what is the current context:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
contexts:
- context:
    cluster: lithe-cocoa-92103_kubernetes
    user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
current-context: lithe-cocoa-92103_kubernetes
kind: Config
preferences: {}
users:
- name: lithe-cocoa-92103_kubernetes
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b
- name: lithe-cocoa-92103_kubernetes-basic-auth
  user:
    password: h5M0FtUUIflBSdI7
    username: admin

$ kubectl config current-context
lithe-cocoa-92103_kubernetes
```

The next step is to define a context for the kubectl client to work in each namespace. The values of “cluster” and “user” fields are copied from the current context.

```
$ kubectl config set-context dev --namespace=development --cluster=lithe-cocoa-921
$ kubectl config set-context prod --namespace=production --cluster=lithe-cocoa-921
```

The above commands provided two request contexts you can alternate against depending on what namespace you wish to work against.

Let’s switch to operate in the development namespace.

```
$ kubectl config use-context dev
```

You can verify your current context by doing the following:

```
$ kubectl config current-context
dev
```

At this point, all requests we make to the Kubernetes cluster from the command line are scoped to the development namespace.

Let’s create some contents.

```
$ kubectl run snowflake --image=kubernetes/serve_hostname --replicas=2
```

We have just created a deployment whose replica size is 2 that is running the pod called snowflake with a basic container that just serves the hostname. Note that `kubectl run` creates deployments only on Kubernetes cluster >= v1.2. If you are running older versions, it creates replication controllers instead. If you want to obtain the old behavior, use `--generator=run/v1` to create replication controllers. See `kubectl run` for more details.

```
$ kubectl get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
snowflake   2         2         2            2           2m

$ kubectl get pods -l run=snowflake
NAME                         READY     STATUS    RESTARTS   AGE
snowflake-3968820950-9dgr8   1/1       Running   0          2m
snowflake-3968820950-vgc4n   1/1       Running   0          2m
```

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the production namespace.

Let’s switch to the production namespace and show how resources in one namespace are hidden from the other.

```
$ kubectl config use-context prod
```

The production namespace should be empty, and the following commands should return nothing.

```
$ kubectl get deployment
$ kubectl get pods
```

Production likes to run cattle, so let's create some cattle pods.

```
$ kubectl run cattle --image=kubernetes/serve_hostname --replicas=5

$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
cattle    5         5         5            5           10s

kubectl get pods -l run=cattle
NAME                        READY     STATUS    RESTARTS   AGE
cattle-2263376956-41xy6     1/1       Running   0          34s
cattle-2263376956-kw466     1/1       Running   0          34s
cattle-2263376956-n4v97     1/1       Running   0          34s
cattle-2263376956-p5p3i     1/1       Running   0          34s
cattle-2263376956-sxpth     1/1       Running   0          34s
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

Understanding the motivation for using namespaces

A single cluster should be able to satisfy the needs of multiple users or groups of users (henceforth a 'user community').

Kubernetes *namespaces* help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

Each user community wants to be able to work in isolation from other communities.

Each user community has its own:

1. resources (pods, services, replication controllers, etc.)

2. policies (who can or cannot perform actions in their community)
3. constraints (this community is allowed this much quota, etc.)

A cluster operator may create a Namespace for each unique user community.

The Namespace provides a unique scope for:

1. named resources (to avoid basic naming collisions)
2. delegated management authority to trusted users
3. ability to limit community resource consumption

Use cases include:

1. As a cluster operator, I want to support multiple user communities on a single cluster.
2. As a cluster operator, I want to delegate authority to partitions of the cluster to trusted users in those communities.
3. As a cluster operator, I want to limit the amount of resources each community can consume in order to limit the impact to other communities using the cluster.
4. As a cluster user, I want to interact with resources that are pertinent to my user community in isolation of what other user communities are doing on the cluster.

Understanding namespaces and DNS

When you create a Service, it creates a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container just uses `<service-name>` it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

What’s next

- Learn more about setting the namespace preference.
- Learn more about setting the namespace for a request
- See namespaces design.

Namespaces Walkthrough

Kubernetes *namespaces* help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

Step Zero: Prerequisites

This example assumes the following:

1. You have an existing Kubernetes cluster.
2. You have a basic understanding of Kubernetes *Pods*, *Services*, and *Deployments*.

Step One: Understand the default namespace

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

Assuming you have a fresh cluster, you can introspect the available namespace's by doing the following:

```
$ kubectl get namespaces
NAME      STATUS    AGE
default   Active    13m
```

Step Two: Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

Let's imagine a scenario where an organization is using a shared Kubernetes cluster for development and production use cases.

The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: development and production.

Let's create two new namespaces to hold our work.

Use the file `namespace-dev.json` which describes a development namespace:

namespace-dev.json

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "development",
    "labels": {
      "name": "development"
    }
  }
}
```

Create the development namespace using kubectl.

```
$ kubectl create -f docs/admin/namespaces/namespace-dev.json
```

And then let's create the production namespace using kubectl.

```
$ kubectl create -f docs/admin/namespaces/namespace-prod.json
```

To be sure things are right, let's list all of the namespaces in our cluster.

```
$ kubectl get namespaces --show-labels
NAME          STATUS    AGE       LABELS
default       Active    32m       <none>
development   Active    29s       name=development
production    Active    23s       name=production
```

Step Three: Create pods in each namespace

A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster.

Users interacting with one namespace do not see the content in another namespace.

To demonstrate this, let's spin up a simple Deployment and Pods in the development namespace.

We first check what is the current context:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
contexts:
- context:
    cluster: lithe-cocoa-92103_kubernetes
    user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
current-context: lithe-cocoa-92103_kubernetes
kind: Config
preferences: {}
users:
- name: lithe-cocoa-92103_kubernetes
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b
- name: lithe-cocoa-92103_kubernetes-basic-auth
  user:
    password: h5M0FtUUIflBSdI7
    username: admin

$ kubectl config current-context
lithe-cocoa-92103_kubernetes
```

The next step is to define a context for the kubectl client to work in each namespace. The value of "cluster" and "user" fields are copied from the current context.

```
$ kubectl config set-context dev --namespace=development --cluster=lithe-cocoa-921
$ kubectl config set-context prod --namespace=production --cluster=lithe-cocoa-921
```

The above commands provided two request contexts you can alternate against depending on what namespace you wish to work against.

Let's switch to operate in the development namespace.

```
$ kubectl config use-context dev
```

You can verify your current context by doing the following:

```
$ kubectl config current-context
dev
```

At this point, all requests we make to the Kubernetes cluster from the command line are scoped to the development namespace.

Let's create some contents.

```
$ kubectl run snowflake --image=kubernetes/serve_hostname --replicas=2
```

We have just created a deployment whose replica size is 2 that is running the pod called snowflake with a basic container that just serves the hostname. Note that `kubectl run` creates deployments only on Kubernetes cluster >= v1.2. If you are running older versions, it creates replication controllers instead. If you want to obtain the old behavior, use `--generator=run/v1` to create replication controllers. See `kubectl run` for more details.

```
$ kubectl get deployment
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
snowflake   2         2         2            2           2m

$ kubectl get pods -l run=snowflake
NAME                         READY     STATUS    RESTARTS   AGE
snowflake-3968820950-9dgr8   1/1       Running   0          2m
snowflake-3968820950-vgc4n   1/1       Running   0          2m
```

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the production namespace.

Let's switch to the production namespace and show how resources in one namespace are hidden from the other.

```
$ kubectl config use-context prod
```

The production namespace should be empty, and the following commands should return nothing.

```
$ kubectl get deployment
$ kubectl get pods
```

Production likes to run cattle, so let's create some cattle pods.

```
$ kubectl run cattle --image=kubernetes/serve_hostname --replicas=5

$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
cattle    5         5         5            5           10s

kubectl get pods -l run=cattle
NAME                      READY     STATUS    RESTARTS   AGE
cattle-2263376956-41xy6   1/1       Running   0          34s
cattle-2263376956-kw466   1/1       Running   0          34s
cattle-2263376956-n4v97   1/1       Running   0          34s
cattle-2263376956-p5p3i   1/1       Running   0          34s
cattle-2263376956-sxpth   1/1       Running   0          34s
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

Autoscale the DNS Service in a Cluster

This page shows how to enable and configure autoscaling of the DNS service in a Kubernetes cluster.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- Make sure the DNS feature itself is enabled.
- Kubernetes version 1.4.0 or later is recommended.

Determining whether DNS horizontal autoscaling is already enabled

List the Deployments in your cluster in the kube-system namespace:

```
kubectl get deployment --namespace=kube-system
```

The output is similar to this:

```
NAME                  DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
...
kube-dns-autoscaler   1         1         1            1           ...
...
```

If you see "kube-dns-autoscaler" in the output, DNS horizontal autoscaling is already enabled, and you can skip to Tuning autoscaling parameters.

Getting the name of your DNS Deployment or ReplicationController

List the Deployments in your cluster in the kube-system namespace:

```
kubectl get deployment --namespace=kube-system
```

The output is similar to this:

```
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
...
kube-dns    1         1         1            1           ...
...
```

In Kubernetes versions earlier than 1.5 DNS is implemented using a ReplicationController instead of a Deployment. So if you don't see kube-dns, or a similar name, in the preceding output, list the ReplicationControllers in your cluster in the kube-system namespace:

```
kubectl get rc --namespace=kube-system
```

The output is similar to this:

```
NAME             DESIRED    CURRENT    READY      AGE
...
kube-dns-v20     1          1          1          ...
...
```

Determining your scale target

If you have a DNS Deployment, your scale target is:

```
Deployment/<your-deployment-name>
```

where is the name of your DNS Deployment. For example, if your DNS Deployment name is kube-dns, your scale target is Deployment/kube-dns.

If you have a DNS ReplicationController, your scale target is:

```
ReplicationController/<your-rc-name>
```

where is the name of your DNS ReplicationController. For example, if your DNS ReplicationController name is kube-dns-v20, your scale target is ReplicationController/kube-dns-v20.

Enabling DNS horizontal autoscaling

In this section, you create a Deployment. The Pods in the Deployment run a container based on the `cluster-proportional-autoscaler-amd64` image.

Create a file named `dns-horizontal-autoscaler.yaml` with this content:

dns-horizontal-autoscaler.yaml

dns-horizontal-autoscaler.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: kube-dns-autoscaler
spec:
  template:
    metadata:
      labels:
        k8s-app: kube-dns-autoscaler
    spec:
      containers:
      - name: autoscaler
        image: gcr.io/google_containers/cluster-proportional-autoscaler-amd64:1.0.
        resources:
            requests:
                cpu: "20m"
                memory: "10Mi"
        command:
          - /cluster-proportional-autoscaler
          - --namespace=kube-system
          - --configmap=kube-dns-autoscaler
          - --target=<SCALE_TARGET>
          # When cluster is using large nodes(with more cores), "coresPerReplica"
          # If using small nodes, "nodesPerReplica" should dominate.
          - --default-params={"linear":{"coresPerReplica":256,"nodesPerReplica":16
          - --logtostderr=true
          - --v=2
```

In the file, replace `<SCALE_TARGET>` with your scale target.

Go to the directory that contains your configuration file, and enter this command to create the Deployment:

```
kubectl create -f dns-horizontal-autoscaler.yaml
```

The output of a successful command is:

```
deployment "kube-dns-autoscaler" created
```

DNS horizontal autoscaling is now enabled.

Tuning autoscaling parameters

Verify that the kube-dns-autoscaler ConfigMap exists:

```
kubectl get configmap --namespace=kube-system
```

The output is similar to this:

```
NAME                   DATA      AGE
...
kube-dns-autoscaler    1         ...
...
```

Modify the data in the ConfigMap:

```
kubectl edit configmap kube-dns-autoscaler --namespace=kube-system
```

Look for this line:

```
linear: '{"coresPerReplica":256,"min":1,"nodesPerReplica":16}'
```

Modify the fields according to your needs. The “min” field indicates the minimal number of DNS backends. The actual number of backends number is calculated using this equation:

```
replicas = max( ceil( cores * 1/coresPerReplica ) , ceil( nodes * 1/nodesPerReplic
```

Note that the values of both `coresPerReplica` and `nodesPerReplica` are integers.

The idea is that when a cluster is using nodes that have many cores, `coresPerReplica` dominates. When a cluster is using nodes that have fewer cores, `nodesPerReplica` dominates.

There are other supported scaling patterns. For details, see cluster-proportional-autoscaler.

Disable DNS horizontal autoscaling

There are a few options for turning DNS horizontal autoscaling. Which option to use depends on different conditions.

Option 1: Scale down the kube-dns-autoscaler deployment to 0 replicas

This option works for all situations. Enter this command:

```
kubectl scale deployment --replicas=0 kube-dns-autoscaler --namespace=kube-system
```

The output is:

```
deployment "kube-dns-autoscaler" scaled
```

Verify that the replica count is zero:

```
kubectl get deployment --namespace=kube-system
```

The output displays 0 in the DESIRED and CURRENT columns:

```
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
...
kube-dns-autoscaler    0         0         0            0           ...
...
```

Option 2: Delete the kube-dns-autoscaler deployment

This option works if kube-dns-autoscaler is under your own control, which means no one will re-create it:

```
kubectl delete deployment kube-dns-autoscaler --namespace=kube-system
```

The output is:

```
deployment "kube-dns-autoscaler" deleted
```

Option 3: Delete the kube-dns-autoscaler manifest file from the master node

This option works if kube-dns-autoscaler is under control of the Addon Manager's control, and you have write access to the master node.

Sign in to the master node and delete the corresponding manifest file. The common path for this kube-dns-autoscaler is:

```
/etc/kubernetes/addons/dns-horizontal-autoscaler/dns-horizontal-autoscaler.yaml
```

After the manifest file is deleted, the Addon Manager will delete the kube-dns-autoscaler Deployment.

Understanding how DNS horizontal autoscaling works

- The cluster-proportional-autoscaler application is deployed separately from the DNS service.
- An autoscaler Pod runs a client that polls the Kubernetes API server for the number of nodes and cores in the cluster.
- A desired replica count is calculated and applied to the DNS backends based on the current schedulable nodes and cores and the given scaling parameters.
- The scaling parameters and data points are provided via a ConfigMap to the autoscaler, and it refreshes its parameters table every poll interval to be up to date with the latest desired scaling parameters.
- Changes to the scaling parameters are allowed without rebuilding or restarting the autoscaler Pod.
- The autoscaler provides a controller interface to support two control patterns: *linear* and *ladder*.

Future enhancements

Control patterns, in addition to linear and ladder, that consider custom metrics are under consideration as a future development.

Scaling of DNS backends based on DNS-specific metrics is under consideration as a future development. The current implementation, which uses the number of nodes and cores in cluster, is limited.

Support for custom metrics, similar to that provided by Horizontal Pod Autoscaling, is under consideration as a future development.

What's next

Learn more about the implementation of cluster-proportional-autoscaler.

Safely Drain a Node while Respecting Application SLOs

This page shows how to safely drain a machine, respecting the application-level disruption SLOs you have specified using PodDisruptionBudget.

Before you begin

This task assumes that you have met the following prerequisites:

- You are using Kubernetes release >= 1.5.
- Either:
  1. You do not require your applications to be highly available during the node drain, or
  2. You have read about the PodDisruptionBudget concept and Configured PodDisruptionBudgets for applications that need them.

Use `kubectl drain` to remove a node from service

You can use **`kubectl drain`** to safely evict all of your pods from a node before you perform maintenance on the node (e.g. kernel upgrade, hardware maintenance, etc.). Safe evictions allow the pod's containers to gracefully terminate and will respect the **`PodDisruptionBudgets`** you have specified.

Note: By default `kubectl drain` will ignore certain system pods on the node that cannot be killed; see the kubectl drain documentation for more details.

When `kubectl drain` returns successfully, that indicates that all of the pods (except the ones excluded as described in the previous paragraph) have been safely evicted (respecting the desired graceful termination period, and without violating any application-level disruption SLOs). It is then safe to bring down the node by powering down its physical machine or, if running on a cloud platform, deleting its virtual machine.

First, identify the name of the node you wish to drain. You can list all of the nodes in your cluster with

```
kubectl get nodes
```

Next, tell Kubernetes to drain the node:

```
kubectl drain <node name>
```

Once it returns (without giving an error), you can power down the node (or equivalently, if on a cloud platform, delete the virtual machine backing the node). If you leave the node in the cluster during the maintenance operation, you need to run

```
kubectl uncordon <node name>
```

afterwards to tell Kubernetes that it can resume scheduling new pods onto the node.

Draining multiple nodes in parallel

The `kubectl drain` command should only be issued to a single node at a time. However, you can run multiple `kubectl drain` commands for different node in parallel, in different terminals or in the background. Multiple drain commands running concurrently will still respect the `PodDisruptionBudget` you specify.

For example, if you have a StatefulSet with three replicas and have set a `PodDisruptionBudget` for that set specifying `minAvailable: 2`. `kubectl drain` will only evict a pod from the StatefulSet if all three pods are ready, and if you issue multiple drain commands in parallel, Kubernetes will respect the PodDisruptionBudget and ensure that only one pod is unavailable at any given time. Any drains that would cause the number of ready replicas to fall below the specified budget are blocked.

The Eviction API

If you prefer not to use kubectl drain (such as to avoid calling to an external command, or to get finer control over the pod eviction process), you can also programmatically cause evictions using the eviction API.

You should first be familiar with using Kubernetes language clients.

The eviction subresource of a pod can be thought of as a kind of policy-controlled DELETE operation on the pod itself. To attempt an eviction (perhaps more REST-precisely, to attempt to *create* an eviction), you POST an attempted operation. Here's an example:

```
{
  "apiVersion": "policy/v1beta1",
  "kind": "Eviction",
  "metadata": {
    "name": "quux",
    "namespace": "default"
  }
}
```

You can attempt an eviction using `curl`:

```
$ curl -v -H 'Content-type: application/json' http://127.0.0.1:8080/api/v1/namespa
```

The API can respond in one of three ways:

- If the eviction is granted, then the pod is deleted just as if you had sent a `DELETE` request to the pod's URL and you get back `200 OK`.
- If the current state of affairs wouldn't allow an eviction by the rules set forth in the budget, you get back `429 Too Many Requests`. This is typically used for generic rate limiting of *any*

requests, but here we mean that this request isn't allowed *right now* but it may be allowed later. Currently, callers do not get any `Retry-After` advice, but they may in future versions.

- If there is some kind of misconfiguration, like multiple budgets pointing at the same pod, you will get `500 Internal Server Error`.

For a given eviction request, there are two cases.

- There is no budget that matches this pod. In this case, the server always returns `200 OK`.
- There is at least one budget. In this case, any of the three above responses may apply.

In some cases, an application may reach a broken state where it will never return anything other than 429 or 500. This can happen, for example, if the replacement pod created by the application's controller does not become ready, or if the last pod evicted has a very long termination grace period.

In this case, there are two potential solutions:

- Abort or pause the automated operation. Investigate the reason for the stuck application, and restart the automation.
- After a suitably long wait, `DELETE` the pod instead of using the eviction API.

Kubernetes does not specify what the behavior should be in this case; it is up to the application owners and cluster owners to establish an agreement on behavior in these cases.

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.

Configure Out Of Resource Handling

The `kubelet` needs to preserve node stability when available compute resources are low.

This is especially important when dealing with incompressible resources such as memory or disk.

If either resource is exhausted, the node would become unstable.

Eviction Policy

The `kubelet` can pro-actively monitor for and prevent against total starvation of a compute resource. In those cases, the `kubelet` can pro-actively fail one or more pods in order to reclaim the starved resource. When the `kubelet` fails a pod, it terminates all containers in the pod, and the `PodPhase` is transitioned to `Failed`.

Eviction Signals

The `kubelet` can support the ability to trigger eviction decisions on the signals described in the table below. The value of each signal is described in the description column based on the `kubelet` summary API.

Eviction Signal	Description
`memory.available`	`memory.available` := `node.status.capacity[memory]` - `node.stats.memory.workingSet`
`nodefs.available`	`nodefs.available` := `node.stats.fs.available`
`nodefs.inodesFree`	`nodefs.inodesFree` := `node.stats.fs.inodesFree`
`imagefs.available`	`imagefs.available` := `node.stats.runtime.imagefs.available`
`imagefs.inodesFree`	`imagefs.inodesFree` := `node.stats.runtime.imagefs.inodesFree`

Each of the above signals supports either a literal or percentage based value. The percentage based value is calculated relative to the total capacity associated with each signal.

The value for `memory.available` is derived from the cgroupfs instead of tools like `free -m`. This is important because `free -m` does not work in a container, and if users use the node allocatable feature, out of resource decisions are made local to the end user pod part of the cgroup hierarchy as well as the root node. This script reproduces the same set of steps that the `kubelet` performs to calculate `memory.available`. The `kubelet` excludes inactive_file (i.e. # of bytes of file-backed memory on inactive LRU list) from its calculation as it assumes that memory is reclaimable under pressure.

`kubelet` supports only two filesystem partitions.

1. The `nodefs` filesystem that kubelet uses for volumes, daemon logs, etc.

2. The `imagefs` filesystem that container runtimes uses for storing images and container writable layers.

`imagefs` is optional. `kubelet` auto-discovers these filesystems using cAdvisor. `kubelet` does not care about any other filesystems. Any other types of configurations are not currently supported by the kubelet. For example, it is *not OK* to store volumes and logs in a dedicated `filesystem`.

In future releases, the `kubelet` will deprecate the existing garbage collection support in favor of eviction in response to disk pressure.

Eviction Thresholds

The `kubelet` supports the ability to specify eviction thresholds that trigger the `kubelet` to reclaim resources.

Each threshold is of the following form:

`<eviction-signal><operator><quantity>`

- valid `eviction-signal` tokens as defined above.
- valid `operator` tokens are `<`
- valid `quantity` tokens must match the quantity representation used by Kubernetes
- an eviction threshold can be expressed as a percentage if ends with `%` token.

For example, if a node has `10Gi` of memory, and the desire is to induce eviction if available memory falls below `1Gi`, an eviction threshold can be specified as either of the following (but not both).

- `memory.available<10%`
- `memory.available<1Gi`

Soft Eviction Thresholds

A soft eviction threshold pairs an eviction threshold with a required administrator specified grace period. No action is taken by the `kubelet` to reclaim resources associated with the eviction signal until that grace period has been exceeded. If no grace period is provided, the `kubelet` will error on startup.

In addition, if a soft eviction threshold has been met, an operator can specify a maximum allowed pod termination grace period to use when evicting pods from the node. If specified, the `kubelet` will use the lesser value among the `pod.Spec.TerminationGracePeriodSeconds` and the max allowed grace period. If not specified, the `kubelet` will kill pods immediately with no graceful termination.

To configure soft eviction thresholds, the following flags are supported:

- `eviction-soft` describes a set of eviction thresholds (e.g. `memory.available<1.5Gi`) that if met over a corresponding grace period would trigger a pod eviction.
- `eviction-soft-grace-period` describes a set of eviction grace periods (e.g. `memory.available=1m30s`) that correspond to how long a soft eviction threshold must hold before triggering a pod eviction.
- `eviction-max-pod-grace-period` describes the maximum allowed grace period (in seconds) to use when terminating pods in response to a soft eviction threshold being met.

Hard Eviction Thresholds

A hard eviction threshold has no grace period, and if observed, the `kubelet` will take immediate action to reclaim the associated starved resource. If a hard eviction threshold is met, the `kubelet` will kill the pod immediately with no graceful termination.

To configure hard eviction thresholds, the following flag is supported:

- `eviction-hard` describes a set of eviction thresholds (e.g. `memory.available<1Gi`) that if met would trigger a pod eviction.

The `kubelet` has the following default hard eviction threshold:

- `--eviction-hard=memory.available<100Mi`

Eviction Monitoring Interval

The `kubelet` evaluates eviction thresholds per its configured housekeeping interval.

- `housekeeping-interval` is the interval between container housekeepings.

Node Conditions

The `kubelet` will map one or more eviction signals to a corresponding node condition.

If a hard eviction threshold has been met, or a soft eviction threshold has been met independent of its associated grace period, the `kubelet` will report a condition that reflects the node is under pressure.

The following node conditions are defined that correspond to the specified eviction signal.

Node Condition	Eviction Signal	Description
`MemoryPressure`	`memory.available`	Available memory on the node has satisfied an eviction threshold
`DiskPressure`	`nodefs.available`, `nodefs.inodesFree`, `imagefs.available`, or `imagefs.inodesFree`	Available disk space and inodes on either the node's root filesytem or image filesystem has satisfied an eviction threshold

The `kubelet` will continue to report node status updates at the frequency specified by `--node-status-update-frequency` which defaults to `10s`.

Oscillation of node conditions

If a node is oscillating above and below a soft eviction threshold, but not exceeding its associated grace period, it would cause the corresponding node condition to constantly oscillate between true and false, and could cause poor scheduling decisions as a consequence.

To protect against this oscillation, the following flag is defined to control how long the `kubelet` must wait before transitioning out of a pressure condition.

- `eviction-pressure-transition-period` is the duration for which the `kubelet` has to wait before transitioning out of an eviction pressure condition.

The `kubelet` would ensure that it has not observed an eviction threshold being met for the specified pressure condition for the period specified before toggling the condition back to `false`.

Reclaiming node level resources

If an eviction threshold has been met and the grace period has passed, the `kubelet` will initiate the process of reclaiming the pressured resource until it has observed the signal has gone below its defined threshold.

The `kubelet` attempts to reclaim node level resources prior to evicting end-user pods. If disk pressure is observed, the `kubelet` reclaims node level resources differently if the machine has a dedicated `imagefs` configured for the container runtime.

With Imagefs

If `nodefs` filesystem has met eviction thresholds, `kubelet` will free up disk space in the following order:

1. Delete dead pods/containers

If `imagefs` filesystem has met eviction thresholds, `kubelet` will free up disk space in the following order:

1. Delete all unused images

Without Imagefs

If `nodefs` filesystem has met eviction thresholds, `kubelet` will free up disk space in the following order:

1. Delete dead pods/containers
2. Delete all unused images

Evicting end-user pods

If the `kubelet` is unable to reclaim sufficient resource on the node, it will begin evicting pods.

The `kubelet` ranks pods for eviction as follows:

- by their quality of service.

- by the consumption of the starved compute resource relative to the pods scheduling request.

As a result, pod eviction occurs in the following order:

- `BestEffort` pods that consume the most of the starved resource are failed first.
- `Burstable` pods that consume the greatest amount of the starved resource relative to their request for that resource are killed first. If no pod has exceeded its request, the strategy targets the largest consumer of the starved resource.
- `Guaranteed` pods that consume the greatest amount of the starved resource relative to their request are killed first. If no pod has exceeded its request, the strategy targets the largest consumer of the starved resource.

A `Guaranteed` pod is guaranteed to never be evicted because of another pod's resource consumption. If a system daemon (i.e. `kubelet` , `docker` , `journald` , etc.) is consuming more resources than were reserved via `system-reserved` or `kube-reserved` allocations, and the node only has `Guaranteed` pod(s) remaining, then the node must choose to evict a `Guaranteed` pod in order to preserve node stability, and to limit the impact of the unexpected consumption to other `Guaranteed` pod(s).

Local disk is a `BestEffort` resource. If necessary, `kubelet` will evict pods one at a time to reclaim disk when `DiskPressure` is encountered. The `kubelet` will rank pods by quality of service. If the `kubelet` is responding to `inode` starvation, it will reclaim `inodes` by evicting pods with the lowest quality of service first. If the `kubelet` is responding to lack of available disk, it will rank pods within a quality of service that consumes the largest amount of disk and kill those first.

With Imagefs

If `nodefs` is triggering evictions, `kubelet` will sort pods based on the usage on `nodefs` - local volumes + logs of all its containers.

If `imagefs` is triggering evictions, `kubelet` will sort pods based on the writable layer usage of all its containers.

Without Imagefs

If `nodefs` is triggering evictions, `kubelet` will sort pods based on their total disk usage - local volumes + logs & writable layer of all its containers.

Minimum eviction reclaim

In certain scenarios, eviction of pods could result in reclamation of small amount of resources. This can result in `kubelet` hitting eviction thresholds in repeated successions. In addition to that, eviction of resources like `disk`, is time consuming.

To mitigate these issues, `kubelet` can have a per-resource `minimum-reclaim`. Whenever `kubelet` observes resource pressure, `kubelet` will attempt to reclaim at least `minimum-reclaim` amount of resource below the configured eviction threshold.

For example, with the following configuration:

```
--eviction-hard=memory.available<500Mi,nodefs.available<1Gi,imagefs.available<100G
--eviction-minimum-reclaim="memory.available=0Mi,nodefs.available=500Mi,imagefs.av
```

If an eviction threshold is triggered for `memory.available`, the `kubelet` will work to ensure that `memory.available` is at least `500Mi`. For `nodefs.available`, the `kubelet` will work to ensure that `nodefs.available` is at least `1.5Gi`, and for `imagefs.available` it will work to ensure that `imagefs.available` is at least `102Gi` before no longer reporting pressure on their associated resources.

The default `eviction-minimum-reclaim` is `0` for all resources.

Scheduler

The node will report a condition when a compute resource is under pressure. The scheduler views that condition as a signal to dissuade placing additional pods on the node.

Node Condition	Scheduler Behavior
`MemoryPressure`	No new `BestEffort` pods are scheduled to the node.
`DiskPressure`	No new pods are scheduled to the node.

Node OOM Behavior

If the node experiences a system OOM (out of memory) event prior to the `kubelet` is able to reclaim memory, the node depends on the oom_killer to respond.

The `kubelet` sets a `oom_score_adj` value for each container based on the quality of service for the pod.

Quality of Service	oom_score_adj
`Guaranteed`	-998
`BestEffort`	1000
`Burstable`	min(max(2, 1000 - (1000 * memoryRequestBytes) / machineMemoryCapacityBytes), 999)

If the `kubelet` is unable to reclaim memory prior to a node experiencing system OOM, the `oom_killer` will calculate an `oom_score` based on the percentage of memory it's using on the node, and then add the `oom_score_adj` to get an effective `oom_score` for the container, and then kills the container with the highest score.

The intended behavior should be that containers with the lowest quality of service that are consuming the largest amount of memory relative to the scheduling request should be killed first in order to reclaim memory.

Unlike pod eviction, if a pod container is OOM killed, it may be restarted by the `kubelet` based on its `RestartPolicy`.

Best Practices

Schedulable resources and eviction policies

Let's imagine the following scenario:

- Node memory capacity: `10Gi`

- Operator wants to reserve 10% of memory capacity for system daemons (kernel, `kubelet`, etc.)
- Operator wants to evict pods at 95% memory utilization to reduce thrashing and incidence of system OOM.

To facilitate this scenario, the `kubelet` would be launched as follows:

```
--eviction-hard=memory.available<500Mi
--system-reserved=memory=1.5Gi
```

Implicit in this configuration is the understanding that “System reserved” should include the amount of memory covered by the eviction threshold.

To reach that capacity, either some pod is using more than its request, or the system is using more than `500Mi`.

This configuration will ensure that the scheduler does not place pods on a node that immediately induce memory pressure and trigger eviction assuming those pods use less than their configured request.

DaemonSet

It is never desired for a `kubelet` to evict a pod that was derived from a `DaemonSet` since the pod will immediately be recreated and rescheduled back to the same node.

At the moment, the `kubelet` has no ability to distinguish a pod created from `DaemonSet` versus any other object. If/when that information is available, the `kubelet` could pro-actively filter those pods from the candidate set of pods provided to the eviction strategy.

In general, it is strongly recommended that `DaemonSet` not create `BestEffort` pods to avoid being identified as a candidate pod for eviction. Instead `DaemonSet` should ideally launch `Guaranteed` pods.

Deprecation of existing feature flags to reclaim disk

`kubelet` has been freeing up disk space on demand to keep the node stable.

As disk based eviction matures, the following `kubelet` flags will be marked for deprecation in favor of the simpler configuration supported around eviction.

Existing Flag	New Flag
`--image-gc-high-threshold`	`--eviction-hard` or `eviction-soft`
`--image-gc-low-threshold`	`--eviction-minimum-reclaim`
`--maximum-dead-containers`	deprecated
`--maximum-dead-containers-per-container`	deprecated
`--minimum-container-ttl-duration`	deprecated
`--low-diskspace-threshold-mb`	`--eviction-hard` or `eviction-soft`
`--outofdisk-transition-frequency`	`--eviction-pressure-transition-period`

Known issues

kubelet may not observe memory pressure right away

The `kubelet` currently polls `cAdvisor` to collect memory usage stats at a regular interval. If memory usage increases within that window rapidly, the `kubelet` may not observe `MemoryPressure` fast enough, and the `OOMKiller` will still be invoked. We intend to integrate with the `memcg` notification API in a future release to reduce this latency, and instead have the kernel tell us when a threshold has been crossed immediately.

If you are not trying to achieve extreme utilization, but a sensible measure of overcommit, a viable workaround for this issue is to set eviction thresholds at approximately 75% capacity. This increases the ability of this feature to prevent system OOMs, and promote eviction of workloads so cluster state can rebalance.

kubelet may evict more pods than needed

The pod eviction may evict more pods than needed due to stats collection timing gap. This can be mitigated by adding the ability to get root container stats on an on-demand basis (https://github.com/google/cadvisor/issues/1247) in the future.

How kubelet ranks pods for eviction in response to inode exhaustion

At this time, it is not possible to know how many inodes were consumed by a particular container. If the `kubelet` observes inode exhaustion, it will evict pods by ranking them by quality of service. The following issue has been opened in cadvisor to track per container inode consumption (https://github.com/google/cadvisor/issues/1422) which would allow us to rank pods by inode consumption. For example, this would let us identify a container that created large numbers of 0 byte files, and evict that pod over others.

Reserve Compute Resources for System Daemons

Kubernetes nodes can be scheduled to `Capacity`. Pods can consume all the available capacity on a node by default. This is an issue because nodes typically run quite a few system daemons that power the OS and Kubernetes itself. Unless resources are set aside for these system daemons, pods and system daemons compete for resources and lead to resource starvation issues on the node.

The `kubelet` exposes a feature named `Node Allocatable` that helps to reserve compute resources for system daemons. Kubernetes recommends cluster administrators to configure `Node Allocatable` based on their workload density on each node.

Node Allocatable

```
      Node Capacity
---------------------------
|     kube-reserved       |
|-------------------------|
|     system-reserved     |
|-------------------------|
|    eviction-threshold   |
|-------------------------|
|                         |
|      allocatable        |
|   (available for pods)  |
|                         |
|                         |
---------------------------
```

`Allocatable` on a Kubernetes node is defined as the amount of compute resources that are available for pods. The scheduler does not over-subscribe `Allocatable`. `CPU`, `memory` and `ephemeral-storage` are supported as of now.

Node Allocatable is exposed as part of `v1.Node` object in the API and as part of `kubectl describe node` in the CLI.

Resources can be reserved for two categories of system daemons in the `kubelet`.

Enabling QoS and Pod level cgroups

To properly enforce node allocatable constraints on the node, you must enable the new cgroup hierarchy via the `--cgroups-per-qos` flag. This flag is enabled by default. When enabled, the `kubelet` will parent all end-user pods under a cgroup hierarchy managed by the `kubelet`.

Configuring a cgroup driver

The `kubelet` supports manipulation of the cgroup hierarchy on the host using a cgroup driver. The driver is configured via the `--cgroup-driver` flag.

The supported values are the following:

- `cgroupfs` is the default driver that performs direct manipulation of the cgroup filesystem on the host in order to manage cgroup sandboxes.

- `systemd` is an alternative driver that manages cgroup sandboxes using transient slices for resources that are supported by that init system.

Depending on the configuration of the associated container runtime, operators may have to choose a particular cgroup driver to ensure proper system behavior. For example, if operators use the `systemd` cgroup driver provided by the `docker` runtime, the `kubelet` must be configured to use the `systemd` cgroup driver.

Kube Reserved

- **Kubelet Flag**: `--kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi]`
- **Kubelet Flag**: `--kube-reserved-cgroup=`

`kube-reserved` is meant to capture resource reservation for kubernetes system daemons like the `kubelet`, `container runtime`, `node problem detector`, etc. It is not meant to reserve resources for system daemons that are run as pods. `kube-reserved` is typically a function of `pod density` on the nodes. This performance dashboard exposes `cpu` and `memory` usage profiles of `kubelet` and `docker engine` at multiple levels of pod density. This blog post explains how the dashboard can be interpreted to come up with a suitable `kube-reserved` reservation.

To optionally enforce `kube-reserved` on system daemons, specify the parent control group for kube daemons as the value for `--kube-reserved-cgroup` kubelet flag.

It is recommended that the kubernetes system daemons are placed under a top level control group (`runtime.slice` on systemd machines for example). Each system daemon should ideally run within its own child control group. Refer to this doc for more details on recommended control group hierarchy.

Note that Kubelet **does not** create `--kube-reserved-cgroup` if it doesn't exist. Kubelet will fail if an invalid cgroup is specified.

System Reserved

- **Kubelet Flag**: `--system-reserved=[cpu=100mi][,][memory=100Mi][,][ephemeral-storage=1Gi]`
- **Kubelet Flag**: `--system-reserved-cgroup=`

`system-reserved` is meant to capture resource reservation for OS system daemons like `sshd`, `udev`, etc. `system-reserved` should reserve `memory` for the `kernel` too since `kernel` memory is not accounted to pods in Kubernetes at this time. Reserving resources for user login sessions is also recommended (`user.slice` in systemd world).

To optionally enforce `system-reserved` on system daemons, specify the parent control group for OS system daemons as the value for `--system-reserved-cgroup` kubelet flag.

It is recommended that the OS system daemons are placed under a top level control group (`system.slice` on systemd machines for example).

Note that Kubelet **does not** create `--system-reserved-cgroup` if it doesn't exist. Kubelet will fail if an invalid cgroup is specified.

Eviction Thresholds

- **Kubelet Flag**: `--eviction-hard=[memory.available<500Mi]`

Memory pressure at the node level leads to System OOMs which affects the entire node and all pods running on it. Nodes can go offline temporarily until memory has been reclaimed. To avoid (or reduce the probability of) system OOMs kubelet provides `Out of Resource` management. Evictions are supported for `memory` and `ephemeral-storage` only. By reserving some memory via `--eviction-hard` flag, the `kubelet` attempts to `evict` pods whenever memory availability on the node drops below the reserved value. Hypothetically, if system daemons did not exist on a node, pods cannot use more than `capacity - eviction-hard`. For this reason, resources reserved for evictions are not available for pods.

Enforcing Node Allocatable

- **Kubelet Flag**: `--enforce-node-allocatable=pods[,][system-reserved][,][kube-reserved]`

The scheduler treats `Allocatable` as the available `capacity` for pods.

`kubelet` enforce `Allocatable` across pods by default. Enforcement is performed by evicting pods whenever the overall usage across all pods exceeds `Allocatable`. More details on eviction policy can be found here. This enforcement is controlled by specifying `pods` value to the kubelet flag `--enforce-node-allocatable`.

Optionally, `kubelet` can be made to enforce `kube-reserved` and `system-reserved` by specifying `kube-reserved` & `system-reserved` values in the same flag. Note that to enforce `kube-reserved` or `system-reserved`, `--kube-reserved-cgroup` or `--system-reserved-cgroup` needs to be specified respectively.

General Guidelines

System daemons are expected to be treated similar to `Guaranteed` pods. System daemons can burst within their bounding control groups and this behavior needs to be managed as part of kubernetes deployments. For example, `kubelet` should have its own control group and share `Kube-reserved` resources with the container runtime. However, Kubelet cannot burst and use up all available Node resources if `kube-reserved` is enforced.

Be extra careful while enforcing `system-reserved` reservation since it can lead to critical system services being CPU starved or OOM killed on the node. The recommendation is to enforce `system-reserved` only if a user has profiled their nodes exhaustively to come up with precise estimates and is confident in their ability to recover if any process in that group is oom_killed.

- To begin with enforce `Allocatable` on `pods`.
- Once adequate monitoring and alerting is in place to track kube system daemons, attempt to enforce `kube-reserved` based on usage heuristics.
- If absolutely necessary, enforce `system-reserved` over time.

The resource requirements of kube system daemons may grow over time as more and more features are added. Over time, kubernetes project will attempt to bring down utilization of node system daemons, but that is not a priority as of now. So expect a drop in `Allocatable` capacity in future releases.

Example Scenario

Here is an example to illustrate Node Allocatable computation:

- Node has `32Gi` of `memory`, `16 CPUs` and `100Gi` of `Storage`
- `--kube-reserved` is set to `cpu=1,memory=2Gi,ephemeral-storage=1Gi`
- `--system-reserved` is set to `cpu=500m,memory=1Gi,ephemeral-storage=1Gi`
- `--eviction-hard` is set to `memory.available<500Mi,nodefs.available<10%`

Under this scenario, `Allocatable` will be `14.5 CPUs`, `28.5Gi` of memory and `98Gi` of local storage. Scheduler ensures that the total memory `requests` across all pods on this node does not exceed `28.5Gi` and storage doesn't exceed `88Gi`. Kubelet evicts pods whenever the overall memory usage exceeds across pods exceed `28.5Gi`, or if overall disk usage exceeds `88Gi` If all processes on the node consume as much CPU as they can, pods together cannot consume more than `14.5 CPUs`.

If `kube-reserved` and/or `system-reserved` is not enforced and system daemons exceed their reservation, `kubelet` evicts pods whenever the overall node memory usage is higher than `31.5Gi` or `storage` is greater than `90Gi`

Feature Availability

As of Kubernetes version 1.2, it has been possible to **optionally** specify `kube-reserved` and `system-reserved` reservations. The scheduler switched to using `Allocatable` instead of `Capacity` when available in the same release.

As of Kubernetes version 1.6, `eviction-thresholds` are being considered by computing `Allocatable`. To revert to the old behavior set `--experimental-allocatable-ignore-eviction` kubelet flag to `true`.

As of Kubernetes version 1.6, `kubelet` enforces `Allocatable` on pods using control groups. To revert to the old behavior unset `--enforce-node-allocatable` kubelet flag. Note that unless `--kube-reserved`, or `--system-reserved` or `--eviction-hard` flags have non-default values, `Allocatable` enforcement does not affect existing deployments.

As of Kubernetes version 1.6, `kubelet` launches pods in their own cgroup sandbox in a dedicated part of the cgroup hierarchy it manages. Operators are required to drain their nodes prior to upgrade of the `kubelet` from prior versions in order to ensure pods and their associated containers are launched in the proper part of the cgroup hierarchy.

As of Kubernetes version 1.7, `kubelet` supports specifying `storage` as a resource for `kube-reserved` and `system-reserved`.

Guaranteed Scheduling For Critical Add-On Pods

Overview

In addition to Kubernetes core components like api-server, scheduler, controller-manager running on a master machine there are a number of add-ons which, for various reasons, must run on a regular cluster node (rather than the Kubernetes master). Some of these add-ons are critical to a fully functional cluster, such as Heapster, DNS, and UI. A cluster may stop working properly if a critical add-on is evicted (either manually or as a side effect of another operation like upgrade) and becomes pending (for example when the cluster is highly utilized and either there are other pending pods that schedule into the space vacated by the evicted critical add-on pod or the amount of resources available on the node changed for some other reason).

Rescheduler: guaranteed scheduling of critical add-ons

Rescheduler ensures that critical add-ons are always scheduled (assuming the cluster has enough resources to run the critical add-on pods in the absence of regular pods). If the scheduler determines that no node has enough free resources to run the critical add-on pod given the pods that are already running in the cluster (indicated by critical add-on pod's pod condition PodScheduled set to false, the reason set to Unschedulable) the rescheduler tries to free up space for the add-on by evicting some pods; then the scheduler will schedule the add-on pod.

To avoid situation when another pod is scheduled into the space prepared for the critical add-on, the chosen node gets a temporary taint "CriticalAddonsOnly" before the eviction(s) (see more details).

Each critical add-on has to tolerate it, while the other pods shouldn't tolerate the taint. The taint is removed once the add-on is successfully scheduled.

Warning: currently there is no guarantee which node is chosen and which pods are being killed in order to schedule critical pods, so if rescheduler is enabled your pods might be occasionally killed for this purpose.

Config

Rescheduler should be enabled by default as a static pod. It doesn't have any user facing configuration (component config) or API and can be disabled:

- during cluster setup by setting `ENABLE_RESCHEDULER` flag to `false`
- on running cluster by deleting its manifest from master node (default path `/etc/kubernetes/manifests/rescheduler.manifest`)

Marking add-on as critical

To be critical an add-on has to run in `kube-system` namespace (configurable via flag) and

- have the `scheduler.alpha.kubernetes.io/critical-pod` annotation set to empty string, and
- have the PodSpec's `tolerations` field set to `[{"key":"CriticalAddonsOnly", "operator":"Exists"}]`

The first one marks a pod a critical. The second one is required by Rescheduler algorithm.

Declare Network Policy

This document helps you get started using the Kubernetes NetworkPolicy API to declare network policies that govern how pods communicate with each other.

Before you begin

You'll need to have a Kubernetes cluster in place, with network policy support. There are a number of network providers that support NetworkPolicy, including:

- Calico
- Cilium
- Kube-router
- Romana
- Weave Net

Note: The above list is sorted alphabetically by product name, not by recommendation or preference. This example is valid for a Kubernetes cluster using any of these providers.

Create an `nginx` deployment and expose it via a service

To see how Kubernetes network policy works, start off by creating an `nginx` deployment and exposing it via a service.

```
$ kubectl run nginx --image=nginx --replicas=2
deployment "nginx" created
$ kubectl expose deployment nginx --port=80
service "nginx" exposed
```

This runs two `nginx` pods in the default namespace, and exposes them through a service called `nginx`.

```
$ kubectl get svc,pod
NAME                        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
svc/kubernetes              10.100.0.1     <none>        443/TCP    46m
svc/nginx                   10.100.0.16    <none>        80/TCP     33s

NAME                        READY          STATUS        RESTARTS   AGE
po/nginx-701339712-e0qfq    1/1            Running       0          35s
po/nginx-701339712-o00ef    1/1            Running       0          35s
```

Test the service by accessing it from another pod

You should be able to access the new `nginx` service from other pods. To test, access the service from another pod in the default namespace. Make sure you haven't enabled isolation on the namespace.

Start a busybox container, and use `wget` on the `nginx` service:

```
$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending,

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

Limit access to the `nginx` service

Let’s say you want to limit access to the `nginx` service so that only pods with the label `access: true` can query it. To do that, create a `NetworkPolicy` that allows connections only from those pods:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      run: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

Assign the policy to the service

Use kubectl to create a NetworkPolicy from the above nginx-policy.yaml file:

```
$ kubectl create -f nginx-policy.yaml
networkpolicy "access-nginx" created
```

Test access to the service when access label is not defined

If we attempt to access the nginx Service from a pod without the correct labels, the request will now time out:

```
$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending,

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
wget: download timed out
/ #
```

Define access label and test again

Create a pod with the correct labels, and you'll see that the request is allowed:

```
$ kubectl run busybox --rm -ti --labels="access=true" --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending,

Hit enter for command prompt

/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

Reconfigure a Node's Kubelet in a Live Cluster

FEATURE STATE: `Kubernetes v1.8` alpha

As of Kubernetes 1.8, the new Dynamic Kubelet Configuration feature is available in alpha. This allows you to change the configuration of Kubelets in a live Kubernetes cluster via first-class Kubernetes concepts. Specifically, this feature allows you to configure individual Nodes' Kubelets via ConfigMaps.

Warning: All Kubelet configuration parameters may be changed dynamically, but not all parameters are safe to change dynamically. This feature is intended for system experts who have a strong understanding of how configuration changes will affect behavior. No documentation currently exists which plainly lists "safe to change" fields, but we plan to add it before this feature graduates from alpha.

Before you begin

- A live Kubernetes cluster with both Master and Node at v1.8 or higher must be running, with the `DynamicKubeletConfig` feature gate enabled and the Kubelet's `--dynamic-config-dir` flag set to a writeable directory on the Node. This flag must be set to enable Dynamic Kubelet Configuration.
- The kubectl command-line tool must be also be v1.8 or higher, and must be configured to communicate with the cluster.

Reconfiguring the Kubelet on a Live Node in your Cluster

Basic Workflow Overview

The basic workflow for configuring a Kubelet in a live cluster is as follows:

1. Write a YAML or JSON configuration file containing the Kubelet's configuration.
2. Wrap this file in a ConfigMap and save it to the Kubernetes control plane.
3. Update the Kubelet's correspoinding Node object to use this ConfigMap.

Each Kubelet watches a configuration reference on its respective Node object. When this reference changes, the Kubelet downloads the new configuration and exits. For the feature to work correctly, you must be running a process manager (like systemd) which will restart the Kubelet when it exits. When the Kubelet is restarted, it will begin using the new configuration.

The new configuration completely overrides the old configuration; unspecified fields in the new configuration will receive their canonical default values. Some CLI flags do not have an associated configuration field, and will not be affected by the new configuration. These fields are defined by the KubeletFlags structure, here.

The status of the Node's Kubelet configuration is reported via the `ConfigOK` condition in the Node status. Once you have updated a Node to use the new ConfigMap, you can observe this condition to confirm that the Node is using the intended configuration. A table describing the possible conditions can be found at the end of this article.

This document describes editing Nodes using `kubectl edit`. There are other ways to modify a Node's spec, including `kubectl patch`, for example, which facilitate scripted workflows.

This document only describes a single Node consuming each ConfigMap. Keep in mind that it is also valid for multiple Nodes to consume the same ConfigMap.

Node Authorizer Workarounds

The Node Authorizer does not yet pay attention to which ConfigMaps are assigned to which Nodes. If you currently use the Node authorizer, your Kubelets will not be automatically granted permission to download their respective ConfigMaps.

The temporary workaround used in this document is to manually create the RBAC Roles and RoleBindings for each ConfigMap. The Node Authorizer will be extended before the Dynamic Kubelet Configuration feature graduates from alpha, so doing this in production should never be necessary.

Generating a file that contains the current configuration

The Dynamic Kubelet Configuration feature allows you to provide an override for the entire configuration object, rather than a per-field overlay. This is a simpler model that makes it easier to trace the source of configuration values and debug issues. The compromise, however, is that you must start with knowledge of the existing configuration to ensure that you only change the fields you intend to change.

In the future, the Kubelet will be bootstrapped from a file on disk, and you will simply edit a copy of this file (which, as a best practice, should live in version control) while creating the first Kubelet ConfigMap. Today, however, the Kubelet is still bootstrapped with command-line flags. Fortunately, there is a dirty trick you can use to generate a config file containing a Node's current configuration. The trick involves hitting the Kubelet server's `configz` endpoint via the kubectl proxy. This endpoint, in its current implementation, is intended to be used only as a debugging aid, which is part of why this is a dirty trick. There is ongoing work to improve the endpoint, and in the future this will be a less "dirty" operation. This trick also requires the `jq` command to be installed on your machine, for unpacking and editing the JSON response from the endpoint.

Do the following to generate the file:

1. Pick a Node to reconfigure. We will refer to this Node's name as NODE_NAME.
2. Start the kubectl proxy in the background with `kubectl proxy --port=8001 &`
3. Run the following command to download and unpack the configuration from the configz endpoint:

```
$ export NODE_NAME=the-name-of-the-node-you-are-reconfiguring
$ curl -sSL http://localhost:8001/api/v1/proxy/nodes/${NODE_NAME}/configz | jq '.k
```

Note that we have to manually add the `kind` and `apiVersion` to the downloaded object, as these are not reported by the configz endpoint. This is one of the limitations of the endpoint that is planned to be fixed in the future.

Edit the configuration file

Using your editor of choice, change one of the parameters in the `kubelet_configz_${NODE_NAME}` file from the previous step. A QPS parameter, `eventRecordQPS` for example, is a good candidate.

Push the configuration file to the control plane

Push the edited configuration file to the control plane with the following command:

```
$ kubectl -n kube-system create configmap my-node-config --from-file=kubelet=kubel
```

You should see a response similar to:

```
apiVersion: v1
data:
  kubelet: |
    {...}
kind: ConfigMap
metadata:
  creationTimestamp: 2017-09-14T20:23:33Z
  name: my-node-config-gkt4c2m4b2
  namespace: kube-system
  resourceVersion: "119980"
  selfLink: /api/v1/namespaces/kube-system/configmaps/my-node-config-gkt4c2m4b2
  uid: 946d785e-998a-11e7-a8dd-42010a800006
```

Note that the configuration data must appear under the ConfigMap's `kubelet` key.

We create the ConfigMap in the `kube-system` namespace, which is appropriate because this ConfigMap configures a Kubernetes system component - the Kubelet.

The `--append-hash` option appends a short checksum of the ConfigMap contents to the name. This is convenient for an edit->push workflow, as it will automatically, yet deterministically, generate new names for new ConfigMaps.

We use the `-o yaml` output format so that the name, namespace, and uid are all reported following creation. We will need these in the next step. We will refer to the name as CONFIG_MAP_NAME and the uid as CONFIG_MAP_UID.

Authorize your Node to read the new ConfigMap

Now that you've created a new ConfigMap, you need to authorize your node to read it. First, create a Role for your new ConfigMap with the following commands:

```
$ export CONFIG_MAP_NAME=name-from-previous-output
$ kubectl -n kube-system create role ${CONFIG_MAP_NAME}-reader --verb=get --resour
```

Next, create a RoleBinding to associate your Node with the new Role:

```
$ kubectl -n kube-system create rolebinding ${CONFIG_MAP_NAME}-reader --role=${CON
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Set the Node to use the new configuration

Edit the Node's reference to point to the new ConfigMap with the following command:

```
kubectl edit node ${NODE_NAME}
```

Once in your editor, add the following YAML under `spec`:

```
configSource:
    configMapRef:
        name: CONFIG_MAP_NAME
        namespace: kube-system
        uid: CONFIG_MAP_UID
```

Be sure to specify all three of `name`, `namespace`, and `uid`.

Observe that the Node begins using the new configuration

Retrieve the Node with `kubectl get node ${NODE_NAME} -o yaml`, and look for the `ConfigOK` condition in `status.conditions`. You should see the message `Using current (UID: CONFIG_MAP_UID)` when the Kubelet starts using the new configuration.

For convenience, you can use the following command (using `jq`) to filter down to the `ConfigOK` condition:

```
$ kubectl get no ${NODE_NAME} -o json | jq '.status.conditions|map(select(.type=="
[
  {
    "lastHeartbeatTime": "2017-09-20T18:08:29Z",
    "lastTransitionTime": "2017-09-20T18:08:17Z",
    "message": "using current (UID: \"2ebc8d1a-9e2a-11e7-a8dd-42010a800006\")",
    "reason": "passing all checks",
    "status": "True",
    "type": "ConfigOK"
  }
]
```

If something goes wrong, you may see one of several different error conditions, detailed in the Table of ConfigOK Conditions, below. When this happens, you should check the Kubelet's log for more details.

Edit the configuration file again

To change the configuration again, we simply repeat the above workflow. Try editing the `kubelet` file, changing the previously changed parameter to a new value.

Push the newly edited configuration to the control plane

Push the new configuration to the control plane in a new ConfigMap with the following command:

```
$ kubectl create configmap my-node-config --namespace=kube-system --from-file=kube
```

This new ConfigMap will get a new name, as we have changed the contents. We will refer to the new name as NEW_CONFIG_MAP_NAME and the new uid as NEW_CONFIG_MAP_UID.

Authorize your Node to read the new ConfigMap

Now that you've created a new ConfigMap, you need to authorize your node to read it. First, create a Role for your new ConfigMap with the following commands:

```
$ export NEW_CONFIG_MAP_NAME=name-from-previous-output
$ kubectl -n kube-system create role ${NEW_CONFIG_MAP_NAME}-reader --verb=get --re
```

Next, create a RoleBinding to associate your Node with the new Role:

```
$ kubectl -n kube-system create rolebinding ${NEW_CONFIG_MAP_NAME}-reader --role=$
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Configure the Node to use the new configuration

Once more, edit the Node's `spec.configSource` with `kubectl edit node ${NODE_NAME}`. Your new `spec.configSource` should look like the following, with `name` and `uid` substituted as necessary:

```
configSource:
    configMapRef:
        name: NEW_CONFIG_MAP_NAME
        namespace: kube-system
        uid: NEW_CONFIG_MAP_UID
```

Observe that the Kubelet is using the new configuration

Once more, retrieve the Node with `kubectl get node ${NODE_NAME} -o yaml`, and look for the `ConfigOK` condition in `status.conditions`. You should the message `Using current (UID: NEW_CONFIG_MAP_UID)` when the Kubelet starts using the new configuration.

Deauthorize your Node fom reading the old ConfigMap

Once you know your Node is using the new configuration and are confident that the new configuration has not caused any problems, it is a good idea to deauthorize the node from reading the old ConfigMap. Run the following commands to remove the RoleBinding and Role:

```
$ kubectl -n kube-system delete rolebinding ${CONFIG_MAP_NAME}-reader
$ kubectl -n kube-system delete role ${CONFIG_MAP_NAME}-reader
```

Note that this does not necessarily prevent the Node from reverting to the old configuration, as it may locally cache the old ConfigMap for an indefinite period of time.

You may optionally also choose to remove the old ConfigMap:

```
$ kubectl -n kube-system delete configmap ${CONFIG_MAP_NAME}
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Reset the Node to use its local default configuration

Finally, if you wish to reset the Node to use the configuration it was provisioned with, simply edit the Node with `kubectl edit node ${NODE_NAME}` and remove the `spec.configSource` subfield.

Observe that the Node is using its local default configuration

After removing this subfield, you should eventually observe that the ConfigOK condition's message reverts to either `using current (default)` or `using current (init)`, depending on how the Node was provisioned.

Deauthorize your Node fom reading the old ConfigMap

Once you know your Node is using the default configuraiton again, it is a good idea to deauthorize the node from reading the old ConfigMap. Run the following commands to remove the RoleBinding and Role:

```
$ kubectl -n kube-system delete rolebinding ${NEW_CONFIG_MAP_NAME}-reader
$ kubectl -n kube-system delete role ${NEW_CONFIG_MAP_NAME}-reader
```

Note that this does not necessarily prevent the Node from reverting to the old ConfigMap, as it may locally cache the old ConfigMap for an indefinite period of time.

You may optionally also choose to remove the old ConfigMap:

```
$ kubectl -n kube-system delete configmap ${NEW_CONFIG_MAP_NAME}
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Kubectl Patch Example

As mentioned above, there are many ways to change a Node's configSource. Here is an example command that uses `kubectl patch`:

```
kubectl patch node ${NODE_NAME} -p "{\"spec\":{\"configSource\":{\"configMapRef\":
```

Understanding ConfigOK Conditions

The following table describes several of the `ConfigOK` Node conditions you might encounter in a cluster that has Dynamic Kubelet Config enabled. If you observe a condition with `status=False`, you should check the Kubelet log for more error details by searching for the message or reason text.

Possible Messages	Possible Reasons	Status
using current (default)	current is set to the local default, and no init config was provided	True
using current (init)	current is set to the local default, and an init config was provided	True
using current (UID: CURRENT_CONFIG_MAP_UID)	passing all checks	True
using last-known-good (default)	• failed to load current (UID: CURRENT_CONFIG_MAP_UID) • failed to parse current (UID: CURRENT_CONFIG_MAP_UID) • failed to validate current (UID: CURRENT_CONFIG_MAP_UID)	False
using last-known-good (init)	• failed to load current (UID: CURRENT_CONFIG_MAP_UID) • failed to parse current (UID: CURRENT_CONFIG_MAP_UID) • failed to validate current (UID: CURRENT_CONFIG_MAP_UID)	False

using last-known-good (UID: LAST_KNOWN_GOOD_CONFIG_MAP_UID)	• failed to load current (UID: CURRENT_CONFIG_MAP_UID) • failed to parse current (UID: CURRENT_CONFIG_MAP_UID) • failed to validate current (UID: CURRENT_CONFIG_MAP_UID)	False
The reasons in the next column could potentially appear for any of the above messages. This condition indicates that the Kubelet is having trouble reconciling `spec.configSource`, and thus no change to the in-use configuration has occurred. The "failed to sync" reasons are specific to the failure that occurred, and the next column does not necessarily contain all possible failure reasons.	failed to sync, reason: • failed to read Node from informer object cache • failed to reset to local (default or init) config • invalid NodeConfigSource, exactly one subfield must be non-nil, but all were nil • invalid ObjectReference, all of UID, Name, and Namespace must be specified • invalid ObjectReference, UID SOME_UID does not match UID of downloaded ConfigMap SOME_OTHER_UID • failed to determine whether object with UID SOME_UID was already checkpointed • failed to download ConfigMap with name SOME_NAME from namespace SOME_NAMESPACE	False

- failed to save config checkpoint for object with UID SOME_UID
- failed to set current config checkpoint to default
- failed to set current config checkpoint to object with UID SOME_UID

Use Calico for NetworkPolicy

This page shows how to use Calico for NetworkPolicy.

Before you begin

- Install Calico for Kubernetes.

Deploying a cluster using Calico

You can deploy a cluster using Calico for network policy in the default GCE deployment using the following set of commands:

```
export NETWORK_POLICY_PROVIDER=calico
export KUBE_NODE_OS_DISTRIBUTION=debian
curl -sS https://get.k8s.io | bash
```

See the Calico documentation for more options to deploy Calico with Kubernetes.

Understanding Calico components

Deploying a cluster with Calico adds Pods that support Kubernetes NetworkPolicy. These Pods run in the `kube-system` Namespace.

To see this list of Pods run:

```
kubectl get pods --namespace=kube-system
```

You'll see a list of Pods similar to this:

```
NAME                                        READY     STATUS    RESTARTS
calico-node-kubernetes-minion-group-jck6    1/1       Running   0
calico-node-kubernetes-minion-group-k9jy    1/1       Running   0
calico-node-kubernetes-minion-group-szgr    1/1       Running   0
calico-policy-controller-65rw1              1/1       Running   0
...
```

There are two main components to be aware of:

- One `calico-node` Pod runs on each node in your cluster and enforces network policy on the traffic to/from Pods on that machine by configuring iptables.
- The `calico-policy-controller` Pod reads the policy and label information from the Kubernetes API and configures Calico appropriately.

What's next

Once your cluster is running, you can follow the NetworkPolicy getting started guide to try out Kubernetes NetworkPolicy.

Use Cilium for NetworkPolicy

This page shows how to use Cilium for NetworkPolicy.

For background on Cilium, read the Introduction to Cilium.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Deploying Cilium on Minikube for Basic Testing

To get familiar with Cilium easily you can follow the Cilium Kubernetes Getting Started Guide to perform a basic DaemonSet installation of Cilium in minikube.

Installation in a minikube setup uses a simple "all-in-one" YAML file that includes DaemonSet configurations for Cilium and a key-value store (consul) as well as appropriate RBAC settings:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/example
clusterrole "cilium" created
serviceaccount "cilium" created
clusterrolebinding "cilium" created
daemonset "cilium-consul" created
daemonset "cilium" created
```

The remainder of the Getting Started Guide explains how to enforce both L3/L4 (i.e., IP address + port) security policies, as well as L7 (e.g., HTTP) security policies using an example application.

Deploying Cilium for Production Use

For detailed instructions around deploying Cilium for production, see: Cilium Administrator Guide This documentation includes detailed requirements, instructions and example production DaemonSet files.

Understanding Cilium components

Deploying a cluster with Cilium adds Pods to the `kube-system` namespace. To see this list of Pods run:

```
kubectl get pods --namespace=kube-system
```

You'll see a list of Pods similar to this:

```
NAME            DESIRED    CURRENT    READY      NODE-SELECTOR   AGE
cilium          1          1          1          <none>          2m
...
```

There are two main components to be aware of:

- One `cilium` Pod runs on each node in your cluster and enforces network policy on the traffic to/from Pods on that node using Linux BPF.

- For production deployments, Cilium should leverage the key-value store cluster (e.g., etcd) used by Kubernetes, which typically runs on the Kubernetes master nodes. The Cilium Administrator Guide includes an example DaemonSet which can be customized to point to this key-value store cluster. The simple "all-in-one" DaemonSet for minikube requires no such configuration because it automatically deploys a `cilium-consul` Pod to provide a key-value store.

What's next

Once your cluster is running, you can follow the NetworkPolicy getting started guide to try out Kubernetes NetworkPolicy with Cilium. Have fun, and if you have questions, contact us using the Cilium Slack Channel.

Use Kube-router for NetworkPolicy

This page shows how to use Kube-router for NetworkPolicy.

Before you begin

You need to have a Kubernetes cluster running. If you do not already have a cluster, you can create one by using any of the cluster installers like Kops, Bootkube, Kubeadm etc.

Installing Kube-router addon

The Kube-router Addon comes with a Network Policy Controller that watches Kubernetes API server for any NetworkPolicy and pods updated and configures iptables rules and ipsets to allow or block traffic as directed by the policies. Please follow the trying Kube-router with cluster installers guide to install Kube-router addon.

What's next

Once you have installed the Kube-router addon, you can follow the NetworkPolicy getting started guide to try out Kubernetes NetworkPolicy.

Romana for NetworkPolicy

This page shows how to use Romana for NetworkPolicy.

Before you begin

Complete steps 1, 2, and 3 of the kubeadm getting started guide.

Installing Romana with kubeadm

Follow the containerized installation guide for kubeadmin.

Applying network policies

To apply network policies use one of the following:

- Romana network policies.
  - Example of Romana network policy.
- The NetworkPolicy API.

What's next

Once your have installed Romana, you can follow the NetworkPolicy getting started guide to try out Kubernetes NetworkPolicy.

Weave Net for NetworkPolicy

This page shows how to use Weave Net for NetworkPolicy.

Before you begin

Complete steps 1, 2, and 3 of the kubeadm getting started guide.

Installing Weave Net addon

Follow the Integrating Kubernetes via the Addon guide.

The Weave Net Addon for Kubernetes comes with a Network Policy Controller that automatically monitors Kubernetes for any NetworkPolicy annotations on all namespaces and configures `iptables` rules to allow or block traffic as directed by the policies.

What's next

Once you have installed the Weave Net addon, you can follow the NetworkPolicy getting started guide to try out Kubernetes NetworkPolicy.

Change the Reclaim Policy of a PersistentVolume

This page shows how to change the reclaim policy of a Kubernetes PersistentVolume.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Why change reclaim policy of a PersistentVolume

`PersistentVolumes` can have various reclaim policies, including "Retain", "Recycle", and "Delete". For dynamically provisioned `PersistentVolumes`, the default reclaim policy is "Delete". This means that a dynamically provisioned volume is automatically deleted when a user deletes the corresponding `PeristentVolumeClaim`. This automatic behavior might be inappropriate if the volume contains precious data. In that case, it is more appropriate to use the "Retain" policy. With the "Retain" policy, if a user deletes a `PersistentVolumeClaim`, the corresponding `PersistentVolume` is not be deleted. Instead, it is moved to the `Released` phase, where all of its data can be manually recovered.

Changing the reclaim policy of a PersistentVolume

1. List the PersistentVolumes in your cluster:

   ```
   kubectl get pv
   ```

   The output is similar to this:

   ```
    NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOL
    pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete
    pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete
    pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete
   ```

   This list also includes the name of the claims that are bound to each volume for easier identification of dynamically provisioned volumes.

2. Choose one of your PersistentVolumes and change its reclaim policy:

   ```
   kubectl patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"
   ```

   where `<your-pv-name>` is the name of your chosen PersistentVolume.

3. Verify that your chosen PersistentVolume has the right policy:

   ```
   kubectl get pv
   ```

   The output is similar to this:

```
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPO
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Retain
```

In the preceding output, you can see that the volume bound to claim `default/claim3` has reclaim policy `Retain`. It will not be automatically deleted when a user deletes claim `default/claim3`.

What's next

- Learn more about PersistentVolumes.
- Learn more about PersistentVolumeClaims.

Reference

- PersistentVolume
- PersistentVolumeClaim
- See the `persistentVolumeReclaimPolicy` field of PersistentVolumeSpec.

Limit Storage Consumption

This example demonstrates an easy way to limit the amount of storage consumed in a namespace.

The following resources are used in the demonstration: ResourceQuota, LimitRange, and PersistentVolumeClaim.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes

Scenario: Limiting Storage Consumption

The cluster-admin is operating a cluster on behalf of a user population and the admin wants to control how much storage a single namespace can consume in order to control cost.

The admin would like to limit:

1. The number of persistent volume claims in a namespace
2. The amount of storage each claim can request
3. The amount of cumulative storage the namespace can have

LimitRange to limit requests for storage

Adding a `LimitRange` to a namespace enforces storage request sizes to a minimum and maximum. Storage is requested via `PersistentVolumeClaim`. The admission controller that enforces limit ranges will reject any PVC that is above or below the values set by the admin.

In this example, a PVC requesting 10Gi of storage would be rejected because it exceeds the 2Gi max.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

Minimum storage requests are used when the underlying storage provider requires certain minimums. For example, AWS EBS volumes have a 1Gi minimum requirement.

StorageQuota to limit PVC count and cumulative storage capacity

Admins can limit the number of PVCs in a namespace as well as the cumulative capacity of those PVCs. New PVCs that exceed either maximum value will be rejected.

In this example, a 6th PVC in the namespace would be rejected because it exceeds the maximum count of 5. Alternatively, a 5Gi maximum quota when combined with the 2Gi max limit above, cannot have 3 PVCs where each has 2Gi. That would be 6Gi requested for a namespace capped at 5Gi.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storagequota
spec:
  hard:
    persistentvolumeclaims: "5"
    requests.storage: "5Gi"
```

Summary

A limit range can put a ceiling on how much storage is requested while a resource quota can effectively cap the storage consumed by a namespace through claim counts and cumulative storage capacity. The allows a cluster-admin to plan their cluster's storage budget without risk of any one project going over their allotment.

Change the default StorageClass

This page shows how to change the default Storage Class that is used to provision volumes for PersistentVolumeClaims that have no special requirements.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Why change the default storage class?

Depending on the installation method, your Kubernetes cluster may be deployed with an existing StorageClass that is marked as default. This default StorageClass is then used to dynamically provision storage for PersistentVolumeClaims that do not require any specific storage class. See PersistentVolumeClaim documentation for details.

The pre-installed default StorageClass may not fit well with your expected workload; for example, it might provision storage that is too expensive. If this is the case, you can either change the default StorageClass or disable it completely to avoid dynamic provisioning of storage.

Simply deleting the default StorageClass may not work, as it may be re-created automatically by the addon manager running in your cluster. Please consult the docs for your installation for details about addon manager and how to disable individual addons.

Changing the default StorageClass

1. List the StorageClasses in your cluster:

   ```
   kubectl get storageclass
   ```

   The output is similar to this:

   ```
    NAME                 TYPE
    standard (default)   kubernetes.io/gce-pd
    gold                 kubernetes.io/gce-pd
   ```

   The default StorageClass is marked by `(default)`.

2. Mark the default StorageClass as non-default:

   The default StorageClass has an annotation `storageclass.kubernetes.io/is-default-class` set to `true`. Any other value or absence of the annotation is interpreted as `false`.

   To mark a StorageClass as non-default, you need to change its value to `false`:

   ```
   kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{
   ```

   where `<your-class-name>` is the name of your chosen StorageClass.

3. Mark a StorageClass as default:

   Similarly to the previous step, you need to add/set the annotation `storageclass.kubernetes.io/is-default-class=true`.

   ```
   kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{
   ```

Please note that at most one StorageClass can be marked as default. If two or more of them are marked as default, Kubernetes ignores the annotation, i.e. it behaves as if there is no default StorageClass.

4. Verify that your chosen StorageClass is default:

```
kubectl get storageclass
```

The output is similar to this:

```
NAME              TYPE
standard          kubernetes.io/gce-pd
gold (default)    kubernetes.io/gce-pd
```

What's next

- Learn more about StorageClasses.

Kubernetes Cloud Controller Manager

Cloud Controller Manager is an alpha feature in 1.8. In upcoming releases it will be the preferred way to integrate Kubernetes with any cloud. This will ensure cloud providers can develop their features independantly from the core Kubernetes release cycles.

Cloud Controller Manager

Kubernetes v1.6 contains a new binary called `cloud-controller-manager`. `cloud-controller-manager` is a daemon that embeds cloud-specific control loops. These cloud-specific control loops were originally in the `kube-controller-manager`. Since cloud providers develop and release at a different pace compared to the Kubernetes project, abstracting the provider-specific code to the `cloud-controller-manager` binary allows cloud vendors to evolve independently from the core Kubernetes code.

The `cloud-controller-manager` can be linked to any cloud provider that satisifies cloudprovider.Interface. For backwards compatibility, the cloud-controller-manager provided in the core Kubernetes project uses the same cloud libraries as `kube-controller-manager`. Cloud providers already supported in Kubernetes core are expected to use the in-tree cloud-controller-manager to transition out of Kubernetes core. In future Kubernetes releases, all cloud controller managers will be developed outside of the core Kubernetes project managed by sig leads or cloud vendors.

Administration

Requirements

Every cloud has their own set of requirements for running their own cloud provider integration, it should not be too different from the requirements when running `kube-controller-manager`. As a general rule of thumb you'll need:

- cloud authentication/authorization: your cloud may require a token or IAM rules to allow access to their APIs
- kubernetes authentication/authorization: cloud-controller-manager may need RBAC rules set to speak to the kubernetes apiserver
- high availabilty: like kube-controller-manager, you may want a high available setup for cloud controller manager using leader election (on by default).

Running cloud-controller-manager

Successfully running cloud-controller-manager requires some changes to your cluster configuration.

- `kube-apiserver` and `kube-controller-manager` MUST NOT specify the `--cloud-provider` flag. This ensures that it does not run any cloud specific loops that would be run by cloud controller manager. In the future, this flag will be deprecated and removed.
- `kubelet` must run with `--cloud-provider=external`. This is to ensure that the kubelet is aware that it must be initialized by the cloud controller manager before it is scheduled any work.
- `kube-apiserver` SHOULD NOT run the `PersistentVolumeLabel` admission controller since the cloud controller manager takes over labeling persistent volumes. To prevent the PersistentVolumeLabel admission plugin from running, make sure the `kube-apiserver` has a `--admission-control` flag with a value that does not include `PersistentVolumeLabel`.
- For the `cloud-controller-manager` to label persistent volumes, initializers will need to be enabled and an InitializerConifguration needs to be added to the system. Follow these instructions to enable initializers. Use the following YAML to create the InitializerConfiguration:

persistent-volume-label-initializer-config.yaml

```yaml
kind: InitializerConfiguration
apiVersion: admissionregistration.k8s.io/v1alpha1
metadata:
  name: pvlabel.kubernetes.io
initializers:
  - name: pvlabel.kubernetes.io
    rules:
    - apiGroups:
      - ""
      apiVersions:
      - "*"
      resources:
      - persistentvolumes
```

Keep in mind that setting up your cluster to use cloud controller manager will change your cluster behaviour in a few ways:

- kubelets specifying `--cloud-provider=external` will add a taint `node.cloudprovider.kubernetes.io/uninitialized` with an effect `NoSchedule` during initialization. This marks the node as needing a second initialization from an external controller before it can be scheduled work. Note that in the event that cloud controller manager is not available, new nodes in the cluster will be left unscheduable. The taint is important since the scheduler may require cloud specific information about nodes such as it's region or type (high cpu, gpu, high memory, spot instance, etc).
- cloud information about nodes in the cluster will no longer be retrieved using local metadata, but instead all API calls to retreive node information will go through cloud controller manager. This may mean you can restrict access to your cloud API on the kubelets for better security. For larger clusters you may want to consider if cloud controller manager will hit rate limits since it is now responsible for almost all API calls to your cloud from within the cluster.

As of v1.8, cloud controller manager can implement:

- node controller - responsible for updating kubernetes nodes using cloud APIs and deleting kubernetes nodes that were deleted on your cloud.

- service controller - responsible for loadbalancers on your cloud against services of type LoadBalancer.
- route controller - responsible for setting up network routes on your cloud
- PersistentVolumeLabel Admission Controller - responsible for labeling persistent volumes on your cloud - ensure that the persistent volume label admission plugin is not enabled on your kube-apiserver.
- any other features you would like to implement if you are running an out-of-tree provider.

Examples

If you are using a cloud that is currently supported in Kubernetes core and would like to adopt cloud controller manager, see the cloud controller manager in kubernetes core.

For cloud controller managers not in Kubernetes core, you can find the respective projects in repos maintained by cloud vendors or sig leads.

- DigitalOcean
- keepalived
- Rancher

For providers already in Kubernetes core, you can run the in-tree cloud controller manager as a Daemonset in your cluster, use the following as a guideline:

cloud-controller-manager-daemonset-example.yaml

```
# This is an example of how to setup cloud-controller-manger as a Daemonset in you
# It assumes that your masters can run pods and has the role node-role.kubernetes.
# Note that this Daemonset will not work straight out of the box for your cloud, t
# meant to be a guideline.

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
```

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
    metadata:
      labels:
        k8s-app: cloud-controller-manager
    spec:
      serviceAccountName: cloud-controller-manager
      containers:
      - name: cloud-controller-manager
        # for in-tree providers we use gcr.io/google_containers/cloud-controller-m
        # this can be replaced with any other image for out-of-tree providers
        image: gcr.io/google_containers/cloud-controller-manager:v1.8.0
        command:
        - /usr/local/bin/cloud-controller-manager
        - --cloud-provider=<YOUR_CLOUD_PROVIDER>   # Add your own cloud provider h
        - --leader-elect=true
        - --use-service-account-credentials
        # these flags will vary for every cloud provider
        - --allocate-node-cidrs=true
        - --configure-cloud-routes=true
        - --cluster-cidr=172.17.0.0/16
      tolerations:
      # this is required so CCM can bootstrap itself
      - key: node.cloudprovider.kubernetes.io/uninitialized
        value: "true"
        effect: NoSchedule
      # this is to have the daemonset runnable on master nodes
      # the taint may vary depending on your cluster setup
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      # this is to restrict CCM to only run on master nodes
      # the node selector may vary depending on your cluster setup
      nodeSelector:
        node-role.kubernetes.io/master: ""
```

Limitations

Running cloud controller manager comes with a few possible limitations. Although these limitations are being addressed in upcoming releases, it's important that you are aware of these limitations for production workloads.

Support for Volumes

Cloud controller manager does not implement any of the volume controllers found in `kube-controller-manager` as the volume integrations also require coordination with kubelets. As we evolve CSI (container storage interface) and add stronger support for flex volume plugins, necessary support will be added to cloud controller manager so that clouds can fully integrate with volumes. Learn more about out-of-tree CSI volume plugins here.

Scalability

In the previous architecture for cloud providers, we relied on kubelets using a local metadata service to retreive node information about itself. With this new architecture, we now fully rely on the cloud controller managers to retrieve information for all nodes. For very larger clusters, you should consider possible bottle necks such as resource requirements and API rate limiting.

Chicken and Egg

The goal of the cloud controller manager project is to decouple development of cloud features from the core Kubernetes project. Unforunately, many aspects of the Kubernetes project has assumptions that cloud provider features are tightly integrated into the project. As a result, adopting this new architecture can create several situations where a request is being made for information from a cloud provider, but the cloud controller manager may not be able to return that information without the original request being complete.

A good example of this is the TLS bootstrapping feature in the Kubelet. Currently, TLS bootstraping assumes that the Kubelet has the ability to ask the cloud provider (or a local metadata service) for all its address types (private, public, etc) but cloud controller manager cannot set a node's address types without being initialzed in the first place which requires that the kubelet has TLS certificates to communicate with the apiserver.

As this initiative evolves, changes will be made to address these issues in upcoming releases.

Developing your own Cloud Controller Manager

To build and develop your own cloud controller manager, read the Developing Cloud Controller Manager doc.

Developing Cloud Controller Manager

Cloud Controller Manager is an alpha feature in 1.8. In upcoming releases it will be the preferred way to integrate Kubernetes with any cloud. This will ensure cloud providers can develop their features independantly from the core Kubernetes release cycles.

Background

Before going into how to build your own cloud controller manager, some background on how it works under the hood is helpful. The cloud controller manager is code from `kube-controller-manager` utilizing Go interfaces to allow implementations from any cloud to be plugged in. Most of the scaffolding and generic controller implementations will be in core, but it will always exec out to the cloud interfaces it is provided, so long as the cloud provider interface is satisifed.

To dive a little deeper into implementation details, all cloud controller managers will import packages from Kubernetes core, the only difference being each project will register their own cloud providers by calling cloudprovider.RegisterCloudProvier where a global variable of available cloud providers is updated.

Developing

Out of Tree

To build an out-of-tree cloud-controller-manager for your cloud, follow these steps:

1. Create a go package with an implementation that satisfies cloudprovider.Interface.

2. Use main.go in cloud-controller-manager from Kubernestes core as a template for your main.go. As mentioned above, the only difference should be the cloud package that will be imported.

3. Import your cloud package in `main.go`, ensure your package has an `init` block to run cloudprovider.RegisterCloudProvider.

Using existing out-of-tree cloud providers as an example may be helpful. You can find the list here.

In Tree

For in-tree cloud providers, you can run the in-tree cloud controller manager as a Daemonset in your cluster. See the running cloud controller manager docs for more details.

Set up High-Availability Kubernetes Masters

Kubernetes version 1.5 adds alpha support for replicating Kubernetes masters in `kube-up` or `kube-down` scripts for Google Compute Engine. This document describes how to use kube-up/down scripts to manage highly available (HA) masters and how HA masters are implemented for use with GCE.

Starting an HA-compatible cluster

To create a new HA-compatible cluster, you must set the following flags in your `kube-up` script:

- `MULTIZONE=true` - to prevent removal of master replicas kubelets from zones different than server's default zone. Required if you want to run master replicas in different zones, which is recommended.
- `ENABLE_ETCD_QUORUM_READS=true` - to ensure that reads from all API servers will return most up-to-date data. If true, reads will be directed to leader etcd replica. Setting this value to true is optional: reads will be more reliable but will also be slower.

Optionally, you can specify a GCE zone where the first master replica is to be created. Set the following flag:

- `KUBE_GCE_ZONE=zone` - zone where the first master replica will run.

The following sample command sets up a HA-compatible cluster in the GCE zone europe-west1-b:

```
$ MULTIZONE=true KUBE_GCE_ZONE=europe-west1-b  ENABLE_ETCD_QUORUM_READS=true ./clu
```

Note that the commands above create a cluster with one master; however, you can add new master replicas to the cluster with subsequent commands.

Adding a new master replica

After you have created an HA-compatible cluster, you can add master replicas to it. You add master replicas by using a `kube-up` script with the following flags:

- `KUBE_REPLICATE_EXISTING_MASTER=true` - to create a replica of an existing master.
- `KUBE_GCE_ZONE=zone` - zone where the master replica will run. Must be in the same region as other replicas' zones.

You don't need to set the `MULTIZONE` or `ENABLE_ETCD_QUORUM_READS` flags, as those are inherited from when you started your HA-compatible cluster.

The following sample command replicates the master on an existing HA-compatible cluster:

```
$ KUBE_GCE_ZONE=europe-west1-c KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-
```

Removing a master replica

You can remove a master replica from an HA cluster by using a `kube-down` script with the following flags:

- `KUBE_DELETE_NODES=false` - to restrain deletion of kubelets.

- `KUBE_GCE_ZONE=zone` - the zone from where master replica will be removed.
- `KUBE_REPLICA_NAME=replica_name` - (optional) the name of master replica to remove. If empty: any replica from the given zone will be removed.

The following sample command removes a master replica from an existing HA cluster:

```
$ KUBE_DELETE_NODES=false KUBE_GCE_ZONE=europe-west1-c ./cluster/kube-down.sh
```

Handling master replica failures

If one of the master replicas in your HA cluster fails, the best practice is to remove the replica from your cluster and add a new replica in the same zone. The following sample commands demonstrate this process:

1. Remove the broken replica:

```
$ KUBE_DELETE_NODES=false KUBE_GCE_ZONE=replica_zone KUBE_REPLICA_NAME=replica_nam
```

2. Add a new replica in place of the old one:

```
$ KUBE_GCE_ZONE=replica-zone KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up
```

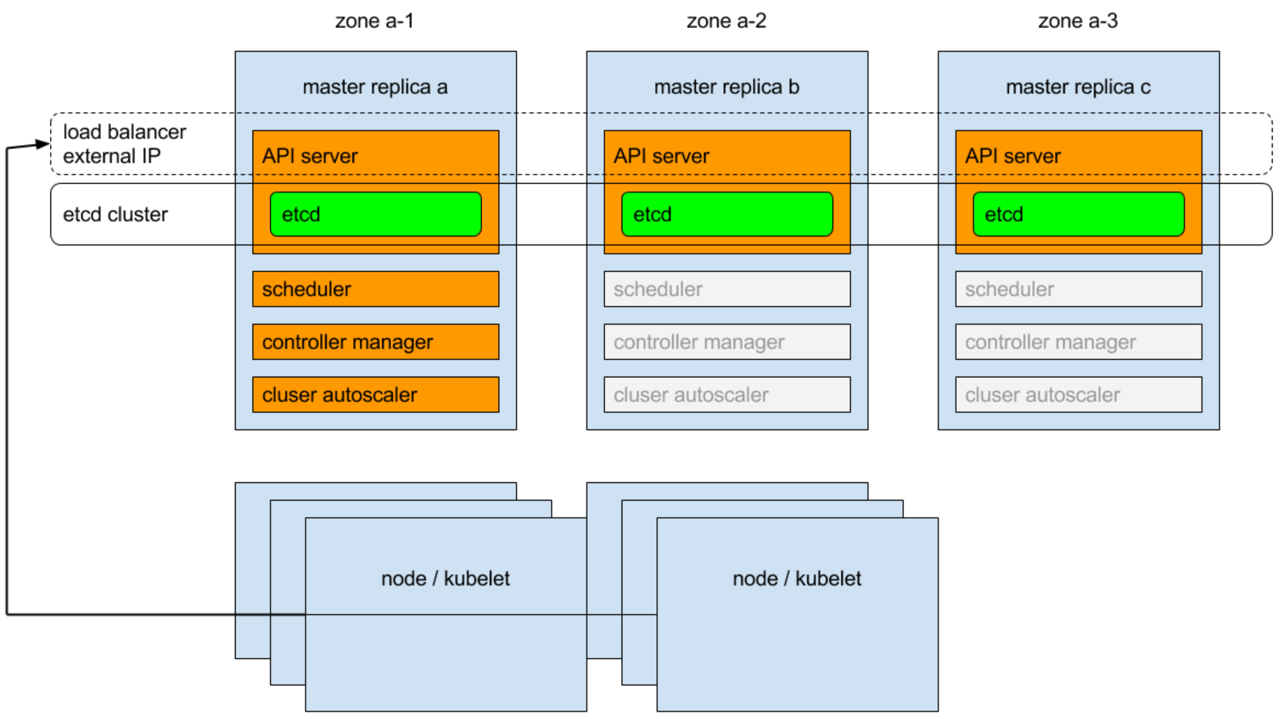
Best practices for replicating masters for HA clusters

- Try to place master replicas in different zones. During a zone failure, all masters placed inside the zone will fail. To survive zone failure, also place nodes in multiple zones (see multiple-zones for details).
- Do not use a cluster with two master replicas. Consensus on a two-replica cluster requires both replicas running when changing persistent state. As a result, both replicas are needed and a

failure of any replica turns cluster into majority failure state. A two-replica cluster is thus inferior, in terms of HA, to a single replica cluster.

- When you add a master replica, cluster state (etcd) is copied to a new instance. If the cluster is large, it may take a long time to duplicate its state. This operation may be sped up by migrating etcd data directory, as described here (we are considering adding support for etcd data dir migration in future).

Implementation notes



Overview

Each of master replicas will run the following components in the following mode:

- etcd instance: all instances will be clustered together using consensus;

- API server: each server will talk to local etcd - all API servers in the cluster will be available;
- controllers, scheduler, and cluster auto-scaler: will use lease mechanism - only one instance of each of them will be active in the cluster;
- add-on manager: each manager will work independently trying to keep add-ons in sync.

In addition, there will be a load balancer in front of API servers that will route external and internal traffic to them.

Load balancing

When starting the second master replica, a load balancer containing the two replicas will be created and the IP address of the first replica will be promoted to IP address of load balancer. Similarly, after removal of the penultimate master replica, the load balancer will be removed and its IP address will be assigned to the last remaining replica. Please note that creation and removal of load balancer are complex operations and it may take some time (~20 minutes) for them to propagate.

Master service & kubelets

Instead of trying to keep an up-to-date list of Kubernetes apiserver in the Kubernetes service, the system directs all traffic to the external IP:

- in one master cluster the IP points to the single master,
- in multi-master cluster the IP points to the load balancer in-front of the masters.

Similarly, the external IP will be used by kubelets to communicate with master.

Master certificates

Kubernetes generates Master TLS certificates for the external public IP and local IP for each replica. There are no certificates for the ephemeral public IP for replicas; to access a replica via its ephemeral public IP, you must skip TLS verification.

Clustering etcd

To allow etcd clustering, ports needed to communicate between etcd instances will be opened (for inside cluster communication). To make such deployment secure, communication between etcd instances is authorized using SSL.

Additional reading

Automated HA master deployment - design doc

Configure Multiple Schedulers

Kubernetes ships with a default scheduler that is described here. If the default scheduler does not suit your needs you can implement your own scheduler. Not just that, you can even run multiple schedulers simultaneously alongside the default scheduler and instruct Kubernetes what scheduler to use for each of your pods. Let's learn how to run multiple schedulers in Kubernetes with an example.

A detailed description of how to implement a scheduler is outside the scope of this document. Please refer to the kube-scheduler implementation in plugin/pkg/scheduler in the Kubernetes source directory for a canonical example.

1. Package the scheduler

Package your scheduler binary into a container image. For the purposes of this example, let's just use the default scheduler (kube-scheduler) as our second scheduler as well. Clone the Kubernetes source code from Github and build the source.

```
git clone https://github.com/kubernetes/kubernetes.git
cd kubernetes
make
```

Create a container image containing the kube-scheduler binary. Here is the `Dockerfile` to build the image:

```
FROM busybox
ADD ./_output/dockerized/bin/linux/amd64/kube-scheduler /usr/local/bin/kube-schedu
```

Save the file as `Dockerfile`, build the image and push it to a registry. This example pushes the image to Google Container Registry (GCR). For more details, please read the GCR documentation.

```
docker build -t my-kube-scheduler:1.0 .
gcloud docker -- push gcr.io/my-gcp-project/my-kube-scheduler:1.0
```

2. Define a Kubernetes Deployment for the scheduler

Now that we have our scheduler in a container image, we can just create a pod config for it and run it in our Kubernetes cluster. But instead of creating a pod directly in the cluster, let's use a Deployment for this example. A Deployment manages a Replica Set which in turn manages the pods, thereby making the scheduler resilient to failures. Here is the deployment config. Save it as `my-scheduler.yaml`:

my-scheduler.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: my-scheduler
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      containers:
      - command:
        - /usr/local/bin/kube-scheduler
        - --address=0.0.0.0
        - --leader-elect=false
        - --scheduler-name=my-scheduler
        image: gcr.io/my-gcp-project/my-kube-scheduler:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
          initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10251
        resources:
          requests:
            cpu: '0.1'
        securityContext:
          privileged: false
        volumeMounts: []
      hostNetwork: false
      hostPID: false
      volumes: []
```

An important thing to note here is that the name of the scheduler specified as an argument to the scheduler command in the container spec should be unique. This is the name that is matched against the value of the optional `spec.schedulerName` on pods, to determine whether this scheduler is responsible for scheduling a particular pod.

Please see the kube-scheduler documentation for detailed description of other command line arguments.

3. Run the second scheduler in the cluster

In order to run your scheduler in a Kubernetes cluster, just create the deployment specified in the config above in a Kubernetes cluster:

```
kubectl create -f my-scheduler.yaml
```

Verify that the scheduler pod is running:

```
$ kubectl get pods --namespace=kube-system
NAME                                           READY     STATUS    RESTARTS   AGE
....
my-scheduler-lnf4s-4744f                       1/1       Running   0          2m
...
```

You should see a “Running” my-scheduler pod, in addition to the default kube-scheduler pod in this list.

To run multiple-scheduler with leader election enabled, you must do the following:

First, update the following fields in your YAML file:

- `--leader-elect=true`
- `--lock-object-namespace=lock-object-namespace`
- `--lock-object-name=lock-object-name`

If RBAC is enabled on your cluster, you must update the `system:kube-scheduler` cluster role. Add you scheduler name to the resourceNames of the rule applied for endpoints resources, as in the following example:

```
$ kubectl edit clusterrole system:kube-scheduler - apiVersion:
rbac.authorization.k8s.io/v1 kind: ClusterRole metadata: annotations:
rbac.authorization.kubernetes.io/autoupdate: "true" labels:
kubernetes.io/bootstrapping: rbac-defaults name: system:kube-scheduler rules: -
apiGroups: - "" resourceNames: - kube-scheduler - my-scheduler resources: -
endpoints verbs: - delete - get - patch - update
```

4. Specify schedulers for pods

Now that our second scheduler is running, let's create some pods, and direct them to be scheduled by either the default scheduler or the one we just deployed. In order to schedule a given pod using a specific scheduler, we specify the name of the scheduler in that pod spec. Let's look at three examples.

- Pod spec without any scheduler name

pod1.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: no-annotation
  labels:
    name: multischeduler-example
spec:
  containers:
  - name: pod-with-no-annotation-container
    image: gcr.io/google_containers/pause:2.0
```

When no scheduler name is supplied, the pod is automatically scheduled using the default-scheduler.

Save this file as `pod1.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod1.yaml
```

- Pod spec with `default-scheduler`

pod2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-default-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: default-scheduler
  containers:
  - name: pod-with-default-annotation-container
    image: gcr.io/google_containers/pause:2.0
```

A scheduler is specified by supplying the scheduler name as a value to `spec.schedulerName`. In this case, we supply the name of the default scheduler which is `default-scheduler`.

Save this file as `pod2.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod2.yaml
```

- Pod spec with `my-scheduler`

pod3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: pod-with-second-annotation-container
    image: gcr.io/google_containers/pause:2.0
```

In this case, we specify that this pod should be scheduled using the scheduler that we deployed - `my-scheduler`. Note that the value of `spec.schedulerName` should match the name supplied to the scheduler command as an argument in the deployment config for the scheduler.

Save this file as `pod3.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod3.yaml
```

Verify that all three pods are running.

```
kubectl get pods
```

Verifying that the pods were scheduled using the desired schedulers

In order to make it easier to work through these examples, we did not verify that the pods were actually scheduled using the desired schedulers. We can verify that by changing the order of pod and deployment config submissions above. If we submit all the pod configs to a Kubernetes cluster before submitting the scheduler deployment config, we see that the pod `second-scheduler` remains in “Pending” state forever while the other two pods get scheduled. Once we submit the scheduler deployment config and our new scheduler starts running, the `second-scheduler` pod gets scheduled as well.

Alternatively, one could just look at the “Scheduled” entries in the event logs to verify that the pods were scheduled by the desired schedulers.

```
kubectl get events
```

IP Masquerade Agent User Guide

This page shows how to configure and enable the ip-masq-agent.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Create an ip-masq-agent

To create an ip-masq-agent, run the following kubectl command:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes-incubator/ip-masq-agent/master/ip-masq-agent.yaml
```

You must also apply the appropriate node label to any nodes in your cluster that you want the agent to run on.

```
kubectl label nodes my-node beta.kubernetes.io/masq-agent-ds-ready=true
```

More information can be found in the ip-masq-agent documentation here

In most cases, the default set of rules should be sufficient; however, if this is not the case for your cluster, you can create and apply a ConfigMap to customize the IP ranges that are affected. For example, to allow only 10.0.0.0/8 to be considered by the ip-masq-agent, you can create the following ConfigMap in a file called "config". **Note:** It is important that the file is called config since, by default, that will be used as the key for lookup by the ip-masq-agent:

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
```

Run the following command to add the config map to your cluster:

```
kubectl create configmap ip-masq-agent --from-file=config --namespace=kube-system
```

This will update a file located at */etc/config/ip-masq-agent* which is periodically checked every *resyscInterval* and applied to the cluster node. After the resync interval has expired, you should see the iptables rules reflect your changes:

```
iptables -t nat -L IP-MASQ-AGENT
Chain IP-MASQ-AGENT (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             169.254.0.0/16       /* ip-masq-agent: cl
RETURN     all  --  anywhere             10.0.0.0/8           /* ip-masq-agent: cl
MASQUERADE  all  --  anywhere             anywhere             /* ip-masq-agent: o
```

By default, the link local range (169.254.0.0/16) is also handled by the ip-masq agent, which sets up the appropriate iptables rules. To have the ip-masq-agent ignore link local, you can set *masqLinkLocal* to true in the config map.

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
masqLinkLocal: true
```

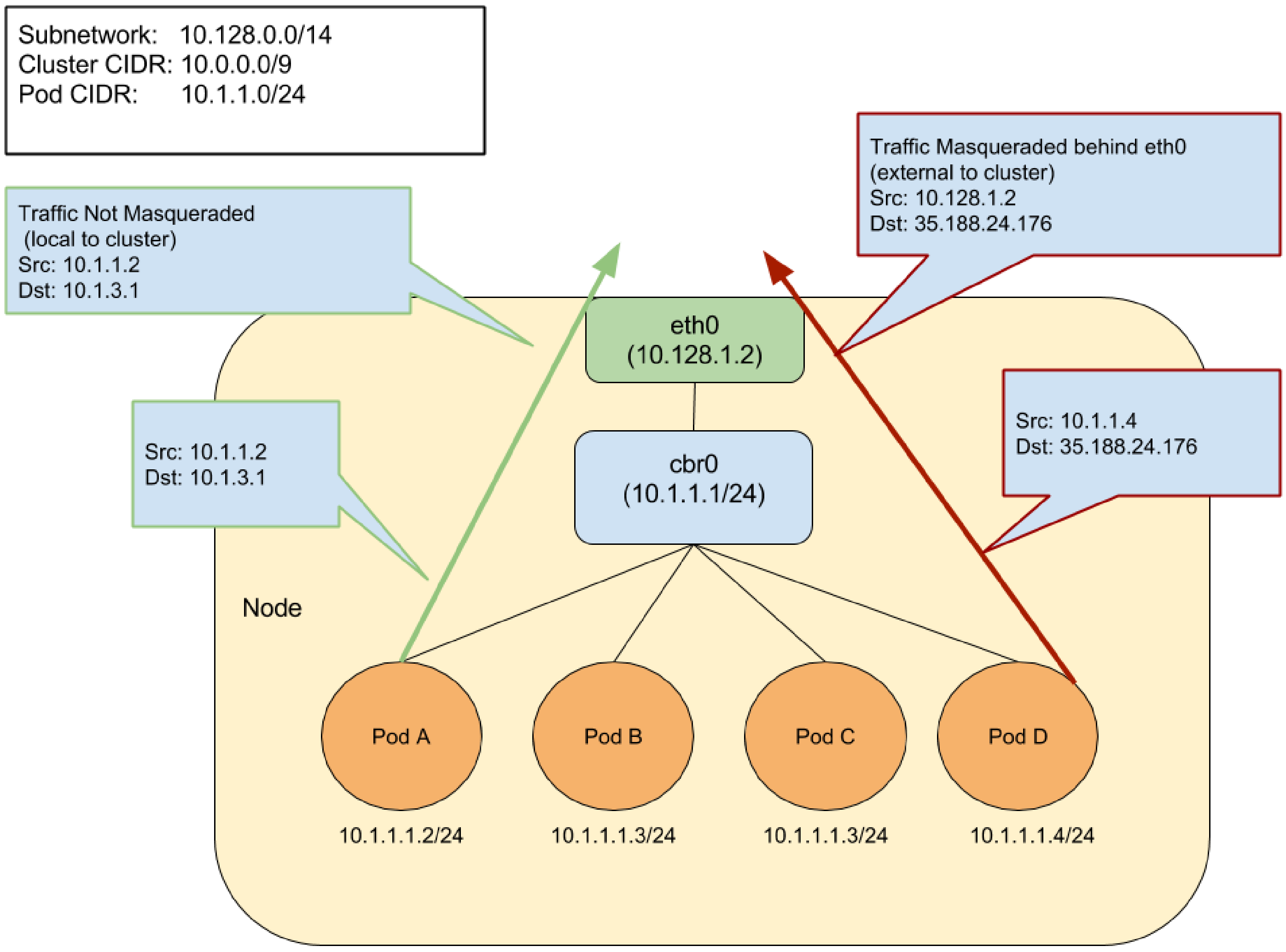
IP Masquerade Agent User Guide

The ip-masq-agent configures iptables rules to hide a pod's IP address behind the cluster node's IP address. This is typically done when sending traffic to destinations outside the cluster's pod CIDR range.

Key Terms

- **NAT (Network Address Translation)** Is a method of remapping one IP address to another by modifying either the source and/or destination address information in the IP header. Typically performed by a device doing IP routing.
- **Masquerading** A form of NAT that is typically used to perform a many to one address translation, where multiple source IP addresses are masked behind a single address, which is typically the device doing the IP routing. In Kubernetes this is the Node's IP address.
- **CIDR (Classless Inter-Domain Routing)** Based on the variable-length subnet masking, allows specifying arbitrary-length prefixes. CIDR introduced a new method of representation for IP addresses, now commonly known as **CIDR notation**, in which an address or routing prefix is written with a suffix indicating the number of bits of the prefix, such as 192.168.2.0/24.
- **Link Local** A link-local address is a network address that is valid only for communications within the network segment or the broadcast domain that the host is connected to. Link-local addresses for IPv4 are defined in the address block 169.254.0.0/16 in CIDR notation.

The ip-masq-agent configures iptables rules to handle masquerading node/pod IP addresses when sending traffic to destinations outside the cluster node's IP and the Cluster IP range. This essentially hides pod IP addresses behind the cluster node's IP address. In some environments, traffic to "external" addresses must come from a known machine address. For example, in Google Cloud, any traffic to the internet must come from a VM's IP. When containers are used, as in GKE, the Pod IP will be rejected for egress. To avoid this, we must hide the Pod IP behind the VM's own IP address - generally known as "masquerade". By default, the agent is configured to treat the three private IP ranges specified by RFC 1918 as non-masquerade CIDR. These ranges are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. The agent will also treat link-local (169.254.0.0/16) as a non-masquerade CIDR by default. The agent is configured to reload its configuration from the location */etc/config/ip-masq-agent* every 60 seconds, which is also configurable.


Subnetwork: 10.128.0.0/14
Cluster CIDR: 10.0.0.0/9
Pod CIDR: 10.1.1.0/24
Traffic Not Masqueraded (local to cluster) Src: 10.1.1.2 Dst: 10.1.3.1
Traffic Masqueraded behind eth0 (external to cluster) Src: 10.128.1.2 Dst: 35.188.24.176
eth0 (10.128.1.2)
cbr0 (10.1.1.1/24)
Src: 10.1.1.2 Dst: 10.1.3.1
Src: 10.1.1.4 Dst: 35.188.24.176
Node
Pod A
Pod B
Pod C
Pod D
10.1.1.1.2/24
10.1.1.1.3/24
10.1.1.1.3/24
10.1.1.1.4/24

The agent configuration file must be written in YAML or JSON syntax, and may contain three optional keys:

- **nonMasqueradeCIDRs:** A list of strings in CIDR notation that specify the non-masquerade ranges.
- **masqLinkLocal:** A Boolean (true / false) which indicates whether to masquerade traffic to the link local prefix 169.254.0.0/16. False by default.
- **resyncInterval:** An interval at which the agent attempts to reload config from disk. e.g. '30s' where 's' is seconds, 'ms' is milliseconds etc…

Traffic to 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16) ranges will NOT be masqueraded. Any other traffic (assumed to be internet) will be masqueraded. An example of a local destination from a pod could be its Node's IP address as well as another node's address or one of the IP addresses in Cluster's IP range. Any other traffic will be masqueraded by default. The below entries show the default set of rules that are applied by the ip-masq-agent:

```
iptables -t nat -L IP-MASQ-AGENT
RETURN     all  --  anywhere             169.254.0.0/16       /* ip-masq-agent: cl
RETURN     all  --  anywhere             10.0.0.0/8           /* ip-masq-agent: cl
RETURN     all  --  anywhere             172.16.0.0/12        /* ip-masq-agent: cl
RETURN     all  --  anywhere             192.168.0.0/16       /* ip-masq-agent: cl
MASQUERADE  all  --  anywhere             anywhere             /* ip-masq-agent: o
```

By default, in GCE/GKE starting with Kubernetes version 1.7.0, if network policy is enabled or you are using a cluster CIDR not in the 10.0.0.0/8 range, the ip-masq-agent will run in your cluster. If you are running in another environment, you can add the ip-masq-agent DaemonSet to your cluster:

Configure private DNS zones and upstream nameservers in Kubernetes

This page shows how to add custom private DNS zones (stub domains) and upstream nameservers.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- Kubernetes version 1.6 and above.
- The cluster must be configured to use the `kube-dns` addon.

Configure stub-domain and upstream DNS servers

Cluster administrators can specify custom stub domains and upstream nameservers by providing a ConfigMap for kube-dns (`kube-system:kube-dns`).

For example, the following ConfigMap sets up a DNS configuration with a single stub domain and two upstream nameservers.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {“acme.local”: [“1.2.3.4”]}
  upstreamNameservers: |
    [“8.8.8.8”, “8.8.4.4”]
```

As specified, DNS requests with the “.acme.local” suffix are forwarded to a DNS listening at 1.2.3.4. Google Public DNS serves the upstream queries.

The table below describes how queries with certain domain names would map to their destination DNS servers:

Domain name	Server answering the query
kubernetes.default.svc.cluster.local	kube-dns
foo.acme.local	custom DNS (1.2.3.4)
widget.com	upstream DNS (one of 8.8.8.8, 8.8.4.4)

See ConfigMap options for details about the configuration option format.

Understanding name resolution in Kubernetes

DNS policies can be set on a per-pod basis. Currently Kubernetes supports two pod-specific DNS policies: “Default” and “ClusterFirst”. These policies are specified with the `dnsPolicy` flag.

NOTE: “Default” is not the default DNS policy. If `dnsPolicy` is not explicitly specified, then “ClusterFirst” is used.

“Default” DNS Policy

If `dnsPolicy` is set to “Default”, then the name resolution configuration is inherited from the node that the pods run on. Custom upstream nameservers and stub domains cannot be used in conjunction with this policy.

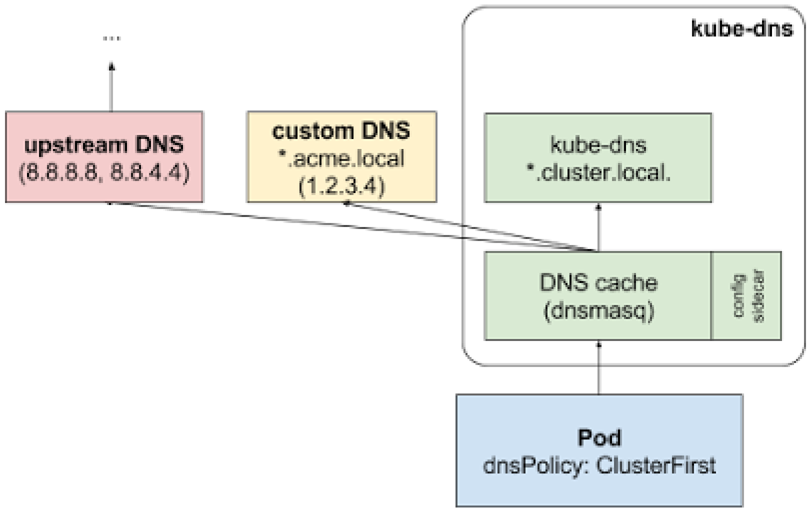
“ClusterFirst” DNS Policy

If the `dnsPolicy` is set to “ClusterFirst”, name resolution is handled differently, *depending on whether stub-domain and upstream DNS servers are configured*.

Without custom configurations: Any query that does not match the configured cluster domain suffix, such as “www.kubernetes.io”, is forwarded to the upstream nameserver inherited from the node.

With custom configurations: If stub domains and upstream DNS servers are configured (as in the previous example), DNS queries will be routed according to the following flow:

1. The query is first sent to the DNS caching layer in kube-dns.
2. From the caching layer, the suffix of the request is examined and then forwarded to the appropriate DNS, based on the following cases:
    1. *Names with the cluster suffix* (e.g.”.cluster.local”): The request is sent to kube-dns.
    2. *Names with the stub domain suffix* (e.g. “.acme.local”): The request is sent to the configured custom DNS resolver (e.g. listening at 1.2.3.4).
    3. *Names without a matching suffix* (e.g.”widget.com”): The request is forwarded to the upstream DNS (e.g. Google public DNS servers at 8.8.8.8 and 8.8.4.4).



ConfigMap options

Options for the kube-dns `kube-system:kube-dns` ConfigMap:

Field	Format	Description
`stubDomains` (optional)	A JSON map using a DNS suffix key (e.g. "acme.local") and a value consisting of a JSON array of DNS IPs.	The target nameserver may itself be a Kubernetes service. For instance, you can run your own copy of dnsmasq to export custom DNS names into the ClusterDNS namespace.
`upstreamNameservers` (optional)	A JSON array of DNS IPs.	Note: If specified, then the values specified replace the nameservers taken by default from the node's `/etc/resolv.conf`. Limits: a maximum of three upstream nameservers can be specified.

Additional examples

Example: Stub domain

In this example, the user has a Consul DNS service discovery system that they wish to integrate with kube-dns. The consul domain server is located at 10.150.0.1, and all consul names have the suffix ".consul.local". To configure Kubernetes, the cluster administrator simply creates a ConfigMap object as shown below.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
    namespace: kube-system
    data:
      stubDomains: |
          {"consul.local": ["10.150.0.1"]}
```

Note that the cluster administrator did not wish to override the node's upstream nameservers, so they did not specify the optional `upstreamNameservers` field.

Example: Upstream nameserver

In this example the cluster administrator wants to explicitly force all non-cluster DNS lookups to go through their own nameserver at 172.16.0.1. Again, this is easy to accomplish; they just need to create a ConfigMap with the `upstreamNameservers` field specifying the desired nameserver.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
    namespace: kube-system
    data:
      upstreamNameservers: |
          [“172.16.0.1”]
```

Cross-cluster Service Discovery using Federated Services

This guide explains how to use Kubernetes Federated Services to deploy a common Service across multiple Kubernetes clusters. This makes it easy to achieve cross-cluster service discovery and availability zone fault tolerance for your Kubernetes applications.

Prerequisites

This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general, and Services in particular.

Overview

Federated Services are created in much that same way as traditional Kubernetes Services by making an API call which specifies the desired properties of your service. In the case of Federated Services, this API call is directed to the Federation API endpoint, rather than a Kubernetes cluster API endpoint. The API for Federated Services is 100% compatible with the API for traditional Kubernetes Services.

Once created, the Federated Service automatically:

1. Creates matching Kubernetes Services in every cluster underlying your Cluster Federation,
2. Monitors the health of those service “shards” (and the clusters in which they reside), and
3. Manages a set of DNS records in a public DNS provider (like Google Cloud DNS, or AWS Route 53), thus ensuring that clients of your federated service can seamlessly locate an appropriate healthy service endpoint at all times, even in the event of cluster, availability zone or regional outages.

Clients inside your federated Kubernetes clusters (i.e. Pods) will automatically find the local shard of the Federated Service in their cluster if it exists and is healthy, or the closest healthy shard in a different cluster if it does not.

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (e.g. Google Cloud, AWS), and on-premises (e.g. on OpenStack). Simply create all of the clusters that you require, in the appropriate cloud providers and/or locations, and register each cluster’s API endpoint and credentials with your Federation API Server (See the federation admin guide for details).

Thereafter, your applications and services can span different clusters and cloud providers as described in more detail below.

Creating a federated service

This is done in the usual way, for example:

```
kubectl --context=federation-cluster create -f services/nginx.yaml
```

The '–context=federation-cluster' flag tells kubectl to submit the request to the Federation API endpoint, with the appropriate credentials. If you have not yet configured such a context, visit the federation admin guide or one of the administration tutorials to find out how to do so.

As described above, the Federated Service will automatically create and maintain matching Kubernetes services in all of the clusters underlying your federation.

You can verify this by checking in each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get services nginx
NAME      CLUSTER-IP     EXTERNAL-IP      PORT(S)   AGE
nginx     10.63.250.98   104.199.136.89   80/TCP    9m
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone. The name and namespace of the underlying services will automatically match those of the Federated Service that you created above (and if you happen to have had services of the same name and namespace already existing in any of those clusters, they will be automatically adopted by the Federation and updated to conform with the specification of your Federated Service - either way, the end result will be the same).

The status of your Federated Service will automatically reflect the real-time status of the underlying Kubernetes services, for example:

```
$kubectl --context=federation-cluster describe services nginx

Name:                   nginx
Namespace:              default
Labels:                 run=nginx
Selector:               run=nginx
Type:                   LoadBalancer
IP:                     10.63.250.98
LoadBalancer Ingress:   104.197.246.190, 130.211.57.243, 104.196.14.231, 104.199.1
Port:                   http    80/TCP
Endpoints:              <none>
Session Affinity:       None
No events.
```

Note the 'LoadBalancer Ingress' addresses of your Federated Service correspond with the 'LoadBalancer Ingress' addresses of all of the underlying Kubernetes services (once these have been allocated - this may take a few seconds). For inter-cluster and inter-cloud-provider networking between service shards to work correctly, your services need to have an externally visible IP address. Service Type: Loadbalancer is typically used for this, although other options (e.g. External IP's) exist.

Note also that we have not yet provisioned any backend Pods to receive the network traffic directed to these addresses (i.e. 'Service Endpoints'), so the Federated Service does not yet consider these to be healthy service shards, and has accordingly not yet added their addresses to the DNS records for this Federated Service (more on this aspect later).

Adding backend pods

To render the underlying service shards healthy, we need to add backend Pods behind them. This is currently done directly against the API endpoints of the underlying clusters (although in future the Federation server will be able to do all this for you with a single command, to save you the trouble). For example, to create backend Pods in 13 underlying clusters:

```
for CLUSTER in asia-east1-c asia-east1-a asia-east1-b \
                        europe-west1-d europe-west1-c europe-west1-b \
                        us-central1-f us-central1-a us-central1-b us-central1-c \
                        us-east1-d us-east1-c us-east1-b
do
  kubectl --context=$CLUSTER run nginx --image=nginx:1.11.1-alpine --port=80
done
```

Note that `kubectl run` automatically adds the `run=nginx` labels required to associate the backend pods with their services.

Verifying public DNS records

Once the above Pods have successfully started and have begun listening for connections, Kubernetes will report them as healthy endpoints of the service in that cluster (via automatic health checks). The Cluster Federation will in turn consider each of these service 'shards' to be healthy, and place them in serving by automatically configuring corresponding public DNS records. You can use your preferred interface to your configured DNS provider to verify this. For example, if your Federation is configured to use Google Cloud DNS, and a managed DNS domain 'example.com':

```
$ gcloud dns managed-zones describe example-dot-com
creationTime: '2016-06-26T18:18:39.229Z'
description: Example domain for Kubernetes Cluster Federation
dnsName: example.com.
id: '3229332181334243121'
kind: dns#managedZone
name: example-dot-com
nameServers:
- ns-cloud-a1.googledomains.com.
- ns-cloud-a2.googledomains.com.
- ns-cloud-a3.googledomains.com.
- ns-cloud-a4.googledomains.com.
```

```
$ gcloud dns record-sets list --zone example-dot-com
NAME                                                            TYPE      TTL
example.com.                                                    NS        21600
example.com.                                                    OA        21600
nginx.mynamespace.myfederation.svc.example.com.                 A         180
nginx.mynamespace.myfederation.svc.us-central1-a.example.com.   A         180
nginx.mynamespace.myfederation.svc.us-central1-b.example.com.   A         180
nginx.mynamespace.myfederation.svc.us-central1-c.example.com.   A         180
nginx.mynamespace.myfederation.svc.us-central1-f.example.com.   CNAME     180
nginx.mynamespace.myfederation.svc.us-central1.example.com.     A         180
nginx.mynamespace.myfederation.svc.asia-east1-a.example.com.    A         180
nginx.mynamespace.myfederation.svc.asia-east1-b.example.com.    CNAME     180
nginx.mynamespace.myfederation.svc.asia-east1-c.example.com.    A         180
nginx.mynamespace.myfederation.svc.asia-east1.example.com.      A         180
nginx.mynamespace.myfederation.svc.europe-west1.example.com.    CNAME     180
nginx.mynamespace.myfederation.svc.europe-west1-d.example.com.  CNAME     180
... etc.
```

Note: If your Federation is configured to use AWS Route53, you can use one of the equivalent AWS tools, for example:

```
$ aws route53 list-hosted-zones
```

and

```
$ aws route53 list-resource-record-sets --hosted-zone-id Z3ECL0L9QLOVBX
```

Whatever DNS provider you use, any DNS query tool (for example 'dig' or 'nslookup') will of course also allow you to see the records created by the Federation for you. Note that you should either point these tools directly at your DNS provider (e.g. `dig @ns-cloud-e1.googledomains.com...`) or expect delays in the order of your configured TTL (180 seconds, by default) before seeing updates, due to caching by intermediate DNS servers.

Some notes about the above example

1. Notice that there is a normal ('A') record for each service shard that has at least one healthy backend endpoint. For example, in us-central1-a, 104.197.247.191 is the external IP address of the service shard in that zone, and in asia-east1-a the address is 130.211.56.221.

2. Similarly, there are regional 'A' records which include all healthy shards in that region. For example, 'us-central1'. These regional records are useful for clients which do not have a particular zone preference, and as a building block for the automated locality and failover mechanism described below.
3. For zones where there are currently no healthy backend endpoints, a CNAME ('Canonical Name') record is used to alias (automatically redirect) those queries to the next closest healthy zone. In the example, the service shard in us-central1-f currently has no healthy backend endpoints (i.e. Pods), so a CNAME record has been created to automatically redirect queries to other shards in that region (us-central1 in this case).
4. Similarly, if no healthy shards exist in the enclosing region, the search progresses further afield. In the europe-west1-d availability zone, there are no healthy backends, so queries are redirected to the broader europe-west1 region (which also has no healthy backends), and onward to the global set of healthy addresses (' nginx.mynamespace.myfederation.svc.example.com.').

The above set of DNS records is automatically kept in sync with the current state of health of all service shards globally by the Federated Service system. DNS resolver libraries (which are invoked by all clients) automatically traverse the hierarchy of 'CNAME' and 'A' records to return the correct set of healthy IP addresses. Clients can then select any one of the returned addresses to initiate a network connection (and fail over automatically to one of the other equivalent addresses if required).

Discovering a federated service

From pods inside your federated clusters

By default, Kubernetes clusters come pre-configured with a cluster-local DNS server ('KubeDNS'), as well as an intelligently constructed DNS search path which together ensure that DNS queries like "myservice", "myservice.mynamespace", "bobsservice.othernamespace" etc issued by your software running inside Pods are automatically expanded and resolved correctly to the appropriate service IP of services running in the local cluster.

With the introduction of Federated Services and Cross-Cluster Service Discovery, this concept is extended to cover Kubernetes services running in any other cluster across your Cluster Federation, globally. To take advantage of this extended range, you use a slightly different DNS name (of the form " ..", e.g. myservice.mynamespace.myfederation) to resolve Federated Services. Using a different DNS name also avoids having your existing applications accidentally traversing cross-zone or cross-region networks and you incurring perhaps unwanted network charges or latency, without you explicitly opting in to this behavior.

So, using our NGINX example service above, and the Federated Service DNS name form just described, let's consider an example: A Pod in a cluster in the `us-central1-f` availability zone needs to contact our NGINX service. Rather than use the service's traditional cluster-local DNS name (`"nginx.mynamespace"` , which is automatically expanded to `"nginx.mynamespace.svc.cluster.local"`) it can now use the service's Federated DNS name, which is `"nginx.mynamespace.myfederation"` . This will be automatically expanded and resolved to the closest healthy shard of my NGINX service, wherever in the world that may be. If a healthy shard exists in the local cluster, that service's cluster-local (typically 10.x.y.z) IP address will be returned (by the cluster-local KubeDNS). This is almost exactly equivalent to non-federated service resolution (almost because KubeDNS actually returns both a CNAME and an A record for local federated services, but applications will be oblivious to this minor technical difference).

But if the service does not exist in the local cluster (or it exists but has no healthy backend pods), the DNS query is automatically expanded to `"nginx.mynamespace.myfederation.svc.us-central1-f.example.com"` (i.e. logically "find the external IP of one of the shards closest to my availability zone"). This expansion is performed automatically by KubeDNS, which returns the associated CNAME record. This results in automatic traversal of the hierarchy of DNS records in the above example, and ends up at one of the external IP's of the Federated Service in the local us-central1 region (i.e. 104.197.247.191, 104.197.244.180 or 104.197.245.170).

It is of course possible to explicitly target service shards in availability zones and regions other than the ones local to a Pod by specifying the appropriate DNS names explicitly, and not relying on automatic DNS expansion. For example, "nginx.mynamespace.myfederation.svc.europe-west1.example.com" will resolve to all of the currently healthy service shards in Europe, even if the Pod issuing the lookup is located in the U.S., and irrespective of whether or not there are healthy shards of the service in the U.S. This is useful for remote monitoring and other similar applications.

From other clients outside your federated clusters

Much of the above discussion applies equally to external clients, except that the automatic DNS expansion described is no longer possible. So external clients need to specify one of the fully qualified DNS names of the Federated Service, be that a zonal, regional or global name. For convenience reasons, it is often a good idea to manually configure additional static CNAME records in your service, for example:

```
eu.nginx.acme.com        CNAME nginx.mynamespace.myfederation.svc.europe-west1.exa
us.nginx.acme.com        CNAME nginx.mynamespace.myfederation.svc.us-central1.exam
nginx.acme.com           CNAME nginx.mynamespace.myfederation.svc.example.com.
```

That way your clients can always use the short form on the left, and always be automatically routed to the closest healthy shard on their home continent. All of the required failover is handled for you automatically by Kubernetes Cluster Federation. Future releases will improve upon this even further.

Handling failures of backend pods and whole clusters

Standard Kubernetes service cluster-IP's already ensure that non-responsive individual Pod endpoints are automatically taken out of service with low latency (a few seconds). In addition, as alluded above, the Kubernetes Cluster Federation system automatically monitors the health of clusters and the endpoints behind all of the shards of your Federated Service, taking shards in and out of service as required (e.g. when all of the endpoints behind a service, or perhaps the entire cluster or availability zone go down, or conversely recover from an outage). Due to the latency inherent in DNS caching (the cache timeout, or TTL for Federated Service DNS records is configured to 3 minutes, by default, but can be adjusted), it may take up to that long for all clients to completely fail over to an alternative cluster in the case of catastrophic failure. However, given the number of discrete IP addresses which can be returned for each regional service endpoint (see e.g. us-central1 above, which has three alternatives) many clients will fail over automatically to one of the alternative IP's in less time than that given appropriate configuration.

Troubleshooting

I cannot connect to my cluster federation API

Check that your

1. Client (typically kubectl) is correctly configured (including API endpoints and login credentials), and
2. Cluster Federation API server is running and network-reachable.

See the federation admin guide to learn how to bring up a cluster federation correctly (or have your cluster administrator do this for you), and how to correctly configure your client.

I can create a federated service successfully against the cluster federation API, but no matching services are created in my underlying clusters

Check that:

1. Your clusters are correctly registered in the Cluster Federation API (`kubectl describe clusters`).
2. Your clusters are all 'Active'. This means that the cluster Federation system was able to connect and authenticate against the clusters' endpoints. If not, consult the logs of the federation-controller-manager pod to ascertain what the failure might be. (`kubectl --namespace=federation logs $(kubectl get pods --namespace=federation -l module=federation-controller-manager -o name`)
3. That the login credentials provided to the Cluster Federation API for the clusters have the correct authorization and quota to create services in the relevant namespace in the clusters. Again you should see associated error messages providing more detail in the above log file if this is not the case.
4. Whether any other error is preventing the service creation operation from succeeding (look for `service-controller` errors in the output of `kubectl logs federation-controller-manager --namespace federation`).

I can create a federated service successfully, but no matching DNS records are created in my DNS provider.

Check that:

1. Your federation name, DNS provider, DNS domain name are configured correctly. Consult the federation admin guide or tutorial to learn how to configure your Cluster Federation system's

DNS provider (or have your cluster administrator do this for you).

2. Confirm that the Cluster Federation's service-controller is successfully connecting to and authenticating against your selected DNS provider (look for `service-controller` errors or successes in the output of `kubectl logs federation-controller-manager --namespace federation`).
3. Confirm that the Cluster Federation's service-controller is successfully creating DNS records in your DNS provider (or outputting errors in its logs explaining in more detail what's failing).

Matching DNS records are created in my DNS provider, but clients are unable to resolve against those names

Check that:

1. The DNS registrar that manages your federation DNS domain has been correctly configured to point to your configured DNS provider's nameservers. See for example Google Domains Documentation and Google Cloud DNS Documentation, or equivalent guidance from your domain registrar and DNS provider.

This troubleshooting guide did not help me solve my problem

1. Please use one of our support channels to seek assistance.

For more information

- Federation proposal details use cases that motivated this work.

Set up Cluster Federation with Kubefed

Kubernetes version 1.5 and above includes a new command line tool called `kubefed` to help you administrate your federated clusters. `kubefed` helps you to deploy a new Kubernetes cluster federation control plane, and to add clusters to or remove clusters from an existing federation control plane.

This guide explains how to administer a Kubernetes Cluster Federation using `kubefed`.

Note: `kubefed` is a beta feature in Kubernetes 1.6.

Prerequisites

This guide assumes that you have a running Kubernetes cluster. Please see one of the getting started guides for installation instructions for your platform.

Getting kubefed

Download the client tarball corresponding to the latest release and extract the binaries in the tarball with the commands:

```
# Linux
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https
tar -xzvf kubernetes-client-linux-amd64.tar.gz

# OS X
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https
tar -xzvf kubernetes-client-darwin-amd64.tar.gz

# Windows
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https
tar -xzvf kubernetes-client-windows-amd64.tar.gz
```

Note: The URLs in the curl commands above download the binaries for `amd64`. If you are on a different architecture, please use a URL appropriate for your architecture. You can find the list of available binaries on the release page.
Copy the extracted binaries to one of the directories in your `$PATH` and set the executable permission on those binaries.

```
sudo cp kubernetes/client/bin/kubefed /usr/local/bin
sudo chmod +x /usr/local/bin/kubefed
sudo cp kubernetes/client/bin/kubectl /usr/local/bin
sudo chmod +x /usr/local/bin/kubectl
```

Install with snap on Ubuntu

kubefed is available as a snap application.

1. If you are on Ubuntu or one of other Linux distributions that support snap package manager, you can install with:

```
sudo snap install kubefed --classic
```

2. Run `kubefed version` to verify that the version you've installed is sufficiently up-to-date.

Choosing a host cluster.

You'll need to choose one of your Kubernetes clusters to be the *host cluster*. The host cluster hosts the components that make up your federation control plane. Ensure that you have a `kubeconfig` entry in your local `kubeconfig` that corresponds to the host cluster. You can verify that you have the required `kubeconfig` entry by running:

```
kubectl config get-contexts
```

The output should contain an entry corresponding to your host cluster, similar to the following:

```
CURRENT   NAME                                          CLUSTER
*         gke_myproject_asia-east1-b_gce-asia-east1     gke_myproject_asia-east1-b
```

You'll need to provide the `kubeconfig` context (called name in the entry above) for your host cluster when you deploy your federation control plane.

Deploying a federation control plane

To deploy a federation control plane on your host cluster, run `kubefed init` command. When you use `kubefed init`, you must provide the following:

- Federation name
- `--host-cluster-context`, the `kubeconfig` context for the host cluster
- `--dns-provider`, one of `'google-clouddns'`, `aws-route53` or `coredns`
- `--dns-zone-name`, a domain name suffix for your federated services

If your host cluster is running in a non-cloud environment or an environment that doesn't support common cloud primitives such as load balancers, you might need additional flags. Please see the on-premises host clusters section below.

The following example command deploys a federation control plane with the name `fellowship`, a host cluster context `rivendell`, and the domain suffix `example.com.`:

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com."
```

The domain suffix specified in `--dns-zone-name` must be an existing domain that you control, and that is programmable by your DNS provider. It must also end with a trailing dot.

Once the federation control plane is initialized, query the namespaces:

```
kubectl get namespace --context=fellowship
```

If you do not see the `default` namespace listed (this is due to a bug). Create it yourself with the following command:

```
kubectl create namespace default --context=fellowship
```

The machines in your host cluster must have the appropriate permissions to program the DNS service that you are using. For example, if your cluster is running on Google Compute Engine, you must enable the Google Cloud DNS API for your project.

The machines in Google Container Engine (GKE) clusters are created without the Google Cloud DNS API scope by default. If you want to use a GKE cluster as a Federation host, you must create it using the `gcloud` command with the appropriate value in the `--scopes` field. You cannot modify a GKE cluster directly to add this scope, but you can create a new node pool for your cluster and delete the old one. *Note that this will cause pods in the cluster to be rescheduled.*

To add the new node pool, run:

```
scopes="$(gcloud container node-pools describe --cluster=gke-cluster default-pool
gcloud container node-pools create new-np \
    --cluster=gke-cluster \
    --scopes="${scopes},https://www.googleapis.com/auth/ndev.clouddns.readwrite"
```

To delete the old node pool, run:

```
gcloud container node-pools delete default-pool --cluster gke-cluster
```

`kubefed init` sets up the federation control plane in the host cluster and also adds an entry for the federation API server in your local kubeconfig. Note that in the beta release in Kubernetes 1.6, `kubefed init` does not automatically set the current context to the newly deployed federation. You can set the current context manually by running:

```
kubectl config use-context fellowship
```

where `fellowship` is the name of your federation.

Basic and token authentication support

`kubefed init` by default only generates TLS certificates and keys to authenticate with the federation API server and writes them to your local kubeconfig file. If you wish to enable basic authentication or token authentication for debugging purposes, you can enable them by passing the `--apiserver-enable-basic-auth` flag or the `--apiserver-enable-token-auth` flag.

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com." \
    --apiserver-enable-basic-auth=true \
    --apiserver-enable-token-auth=true
```

Passing command line arguments to federation components

`kubefed init` bootstraps a federation control plane with default arguments to federation API server and federation controller manager. Some of these arguments are derived from `kubefed init`'s flags. However, you can override these command line arguments by passing them via the appropriate override flags.

You can override the federation API server arguments by passing them to `--apiserver-arg-overrides` and override the federation controller manager arguments by passing them to `--controllermanager-arg-overrides`.

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com." \
    --apiserver-arg-overrides="--anonymous-auth=false,--v=4" \
    --controllermanager-arg-overrides="--controllers=services=false"
```

Configuring a DNS provider

The Federated service controller programs a DNS provider to expose federated services via DNS names. Certain cloud providers automatically provide the configuration required to program the DNS provider if the host cluster's cloud provider is same as the DNS provider. In all other cases, you have to provide the DNS provider configuration to your federation controller manager which will in-turn be passed to the federated service controller. You can provide this configuration to federation controller manager by storing it in a file and passing the file's local filesystem path to `kubefed init`'s `--dns-provider-config` flag. For example, save the config below in `$HOME/coredns-provider.conf`.

```
[Global]
etcd-endpoints = http://etcd-cluster.ns:2379
zones = example.com.
```

And then pass this file to `kubefed init`:

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="coredns" \
    --dns-zone-name="example.com." \
    --dns-provider-config="$HOME/coredns-provider.conf"
```

On-premises host clusters

API server service type

`kubefed init` exposes the federation API server as a Kubernetes service on the host cluster. By default, this service is exposed as a load balanced service. Most on-premises and bare-metal environments, and some cloud environments lack support for load balanced services. `kubefed init` allows exposing the federation API server as a `NodePort` service on such environments. This can be accomplished by passing the `--api-server-service-type=NodePort` flag. You can also specify the preferred address to advertise the federation API server by passing the `--api-server-advertise-address=<IP-address>` flag. Otherwise, one of the host cluster's node address is chosen as the default.

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com." \
    --api-server-service-type="NodePort" \
    --api-server-advertise-address="10.0.10.20"
```

Provisioning storage for etcd

Federation control plane stores its state in `etcd`. `etcd` data must be stored in a persistent storage volume to ensure correct operation across federation control plane restarts. On host clusters that support dynamic provisioning of storage volumes, `kubefed init` dynamically provisions a `PersistentVolume` and binds it to a `PersistentVolumeClaim` to store `etcd` data. If your host cluster doesn't support dynamic provisioning, you can also statically provision a `PersistentVolume`. `kubefed init` creates a `PersistentVolumeClaim` that has the following configuration:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.alpha.kubernetes.io/storage-class: "yes"
  labels:
    app: federated-cluster
  name: fellowship-federation-apiserver-etcd-claim
  namespace: federation-system
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

To statically provision a `PersistentVolume`, you must ensure that the `PersistentVolume` that you create has the matching storage class, access mode and at least as much capacity as the requested `PersistentVolumeClaim`.

Alternatively, you can disable persistent storage completely by passing `--etcd-persistent-storage=false` to `kubefed init`. However, we do not recommended this because your federation control plane cannot survive restarts in this mode.

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com." \
    --etcd-persistent-storage=false
```

`kubefed init` still doesn’t support attaching an existing `PersistentVolumeClaim` to the federation control plane that it bootstraps. We are planning to support this in a future version of `kubefed`.

CoreDNS support

Federated services now support CoreDNS as one of the DNS providers. If you are running your clusters and federation in an environment that does not have access to cloud-based DNS providers, then you can run your own CoreDNS instance and publish the federated service DNS names to that server.

You can configure your federation to use CoreDNS, by passing appropriate values to `kubefed init`'s `--dns-provider` and `--dns-provider-config` flags.

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="coredns" \
    --dns-zone-name="example.com." \
    --dns-provider-config="$HOME/coredns-provider.conf"
```

For more information see Setting up CoreDNS as DNS provider for Cluster Federation.

Adding a cluster to a federation

Once you've deployed a federation control plane, you'll need to make that control plane aware of the clusters it should manage. You can add a cluster to your federation by using the `kubefed join` command.

To use `kubefed join`, you'll need to provide the name of the cluster you want to add to the federation, and the `--host-cluster-context` for the federation control plane's host cluster.
Note: The name that you provide to the `join` command is used as the joining cluster's identity in federation. This name should adhere to the rules described in the identifiers doc. If the context corresponding to your joining cluster conforms to these rules then you can use the same name in the join command. Otherwise, you will have to choose a different name for your cluster's identity. For more information, please see the naming rules and customization section below.
The following example command adds the cluster `gondor` to the federation running on host cluster `rivendell`:

```
kubefed join gondor --host-cluster-context=rivendell
```

Note: Kubernetes requires that you manually join clusters to a federation because the federation control plane manages only those clusters that it is responsible for managing. Adding a cluster tells the federation control plane that it is responsible for managing that cluster.

Naming rules and customization

The cluster name you supply to `kubefed join` must be a valid RFC 1035 label and are enumerated in the Identifiers doc.

Furthermore, federation control plane requires credentials of the joined clusters to operate on them. These credentials are obtained from the local kubeconfig. `kubefed join` uses the cluster name specified as the argument to look for the cluster's context in the local kubeconfig. If it fails to find a matching context, it exits with an error.

This might cause issues in cases where context names for each cluster in the federation don't follow RFC 1035 label naming rules. In such cases, you can specify a cluster name that conforms to the RFC 1035 label naming rules and specify the cluster context using the `--cluster-context` flag. For example, if context of the cluster you are joining is `gondor_needs-no_king`, then you can join the cluster by running:

```
kubefed join gondor --host-cluster-context=rivendell --cluster-context=gondor_need
```

Secret name

Cluster credentials required by the federation control plane as described above are stored as a secret in the host cluster. The name of the secret is also derived from the cluster name.

However, the name of a secret object in Kubernetes should conform to the DNS subdomain name specification described in RFC 1123. If this isn't the case, you can pass the secret name to `kubefed join` using the `--secret-name` flag. For example, if the cluster name is `noldor` and the secret name is `11kingdom`, you can join the cluster by running:

```
kubefed join noldor --host-cluster-context=rivendell --secret-name=11kingdom
```

Note: If your cluster name does not conform to the DNS subdomain name specification, all you need to do is supply the secret name via the `--secret-name` flag. `kubefed join` automatically creates the secret for you.

`kube-dns` configuration

`kube-dns` configuration must be updated in each joining cluster to enable federated service discovery. If the joining Kubernetes cluster is version 1.5 or newer and your `kubefed` is version 1.6 or newer, then this configuration is automatically managed for you when the clusters are joined or unjoined using `kubefed join` or `unjoin` commands.

In all other cases, you must update `kube-dns` configuration manually as described in the Updating KubeDNS section of the admin guide.

Removing a cluster from a federation

To remove a cluster from a federation, run the `kubefed unjoin` command with the cluster name and the federation's `--host-cluster-context`:

```
kubefed unjoin gondor --host-cluster-context=rivendell
```

Turning down the federation control plane

Proper cleanup of federation control plane is not fully implemented in this beta release of `kubefed`. However, for the time being, deleting the federation system namespace should remove all the resources except the persistent storage volume dynamically provisioned for the federation control plane's etcd. You can delete the federation namespace by running the following command:

```
$ kubectl delete ns federation-system
```

Set up CoreDNS as DNS provider for Cluster Federation

This page shows how to configure and deploy CoreDNS to be used as the DNS provider for Cluster Federation.

Objectives

- Configure and deploy CoreDNS server
- Bring up federation with CoreDNS as dns provider
- Setup CoreDNS server in nameserver lookup chain

Before you begin

- You need to have a running Kubernetes cluster (which is referenced as host cluster). Please see one of the getting started guides for installation instructions for your platform.
- Support for `LoadBalancer` services in member clusters of federation is mandatory to enable `CoreDNS` for service discovery across federated clusters.

Deploying CoreDNS and etcd charts

CoreDNS can be deployed in various configurations. Explained below is a reference and can be tweaked to suit the needs of the platform and the cluster federation.

To deploy CoreDNS, we shall make use of helm charts. CoreDNS will be deployed with etcd as the backend and should be pre-installed. etcd can also be deployed using helm charts. Shown below are the instructions to deploy etcd.

```
helm install --namespace my-namespace --name etcd-operator stable/etcd-operator
helm upgrade --namespace my-namespace --set cluster.enabled=true etcd-operator sta
```

Note: etcd default deployment configurations can be overridden, suiting the host cluster.

After deployment succeeds, etcd can be accessed with the http://etcd-cluster.my-namespace:2379 endpoint within the host cluster.

The CoreDNS default configuration should be customized to suit the federation. Shown below is the Values.yaml, which overrides the default configuration parameters on the CoreDNS chart.

Values.yaml

```
isClusterService: false
serviceType: "LoadBalancer"
middleware:
  kubernetes:
    enabled: false
  etcd:
    enabled: true
    zones:
    - "example.com."
    endpoint: "http://etcd-cluster.my-namespace:2379"
```

The above configuration file needs some explanation:

- `isClusterService` specifies whether CoreDNS should be deployed as a cluster-service, which is the default. You need to set it to false, so that CoreDNS is deployed as a Kubernetes application service.
- `serviceType` specifies the type of Kubernetes service to be created for CoreDNS. You need to choose either “LoadBalancer” or “NodePort” to make the CoreDNS service accessible outside the Kubernetes cluster.

- Disable `middleware.kubernetes`, which is enabled by default by setting `middleware.kubernetes.enabled` to false.
- Enable `middleware.etcd` by setting `middleware.etcd.enabled` to true.
- Configure the DNS zone (federation domain) for which CoreDNS is authoritative by setting `middleware.etcd.zones` as shown above.
- Configure the etcd endpoint which was deployed earlier by setting `middleware.etcd.endpoint`

Now deploy CoreDNS by running

```
helm install --namespace my-namespace --name coredns -f Values.yaml stable/coredns
```

Verify that both etcd and CoreDNS pods are running as expected.

Deploying Federation with CoreDNS as DNS provider

The Federation control plane can be deployed using `kubefed init`. CoreDNS can be chosen as the DNS provider by specifying two additional parameters.

```
--dns-provider=coredns
--dns-provider-config=coredns-provider.conf
```

coredns-provider.conf has below format:

```
[Global]
etcd-endpoints = http://etcd-cluster.my-namespace:2379
zones = example.com.
coredns-endpoints = <coredns-server-ip>:<port>
```

- `etcd-endpoints` is the endpoint to access etcd.
- `zones` is the federation domain for which CoreDNS is authoritative and is same as –dns-zone-name flag of `kubefed init`.

- `coredns-endpoints` is the endpoint to access CoreDNS server. This is an optional parameter introduced from v1.7 onwards.

Note: middleware.etcd.zones in CoreDNS configuration and –dns-zone-name flag to kubefed init should match.

Setup CoreDNS server in nameserver resolv.conf chain

Note: The following section applies only to versions prior to v1.7 and will be automatically taken care of if the `coredns-endpoints` parameter is configured in `coredns-provider.conf` as described in section above.

Once the federation control plane is deployed and federated clusters are joined to the federation, you need to add the CoreDNS server to the pod's nameserver resolv.conf chain in all the federated clusters as this self hosted CoreDNS server is not discoverable publicly. This can be achieved by adding the below line to `dnsmasq` container's arg in `kube-dns` deployment.

```
--server=/example.com./<CoreDNS endpoint>
```

Replace `example.com` above with federation domain.

Now the federated cluster is ready for cross-cluster service discovery!

Set up placement policies in Federation

This page shows how to enforce policy-based placement decisions over Federated resources using an external policy engine.

Before you begin

You need to have a running Kubernetes cluster (which is referenced as host cluster). Please see one of the getting started guides for installation instructions for your platform.

Deploying Federation and configuring an external policy engine

The Federation control plane can be deployed using `kubefed init`.

After deploying the Federation control plane, you must configure an Admission Controller in the Federation API server that enforces placement decisions received from the external policy engine.

```
kubectl create -f scheduling-policy-admission.yaml
```

Shown below is an example ConfigMap for the Admission Controller:

scheduling-policy-admission.yaml

scheduling-policy-admission.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: admission
  namespace: federation-system
data:
  config.yml: |
    apiVersion: apiserver.k8s.io/v1alpha1
    kind: AdmissionConfiguration
    plugins:
    - name: SchedulingPolicy
      path: /etc/kubernetes/admission/scheduling-policy-config.yml
  scheduling-policy-config.yml: |
    kubeconfig: /etc/kubernetes/admission/opa-kubeconfig
  opa-kubeconfig: |
    clusters:
      - name: opa-api
        cluster:
          server: http://opa.federation-system.svc.cluster.local:8181/v0/data/kube
    users:
      - name: scheduling-policy
        user:
          token: deadbeefsecret
    contexts:
      - name: default
        context:
          cluster: opa-api
          user: scheduling-policy
    current-context: default
```

The ConfigMap contains three files:

- `config.yml` specifies the location of the `SchedulingPolicy` Admission Controller config file.
- `scheduling-policy-config.yml` specifies the location of the kubeconfig file required to contact the external policy engine. This file can also include a `retryBackoff` value that controls the initial retry backoff delay in milliseconds.
- `opa-kubeconfig` is a standard kubeconfig containing the URL and credentials needed to contact the external policy engine.

Edit the Federation API server deployment to enable the `SchedulingPolicy` Admission Controller.

```
kubectl -n federation-system edit deployment federation-apiserver
```

Update the Federation API server command line arguments to enable the Admission Controller and mount the ConfigMap into the container. If there's an existing `--admission-control` flag, append `,SchedulingPolicy` instead of adding another line.

```
--admission-control=SchedulingPolicy
--admission-control-config-file=/etc/kubernetes/admission/config.yml
```

Add the following volume to the Federation API server pod:

```
- name: admission-config
  configMap:
    name: admission
```

Add the following volume mount the Federation API server `apiserver` container:

```
volumeMounts:
- name: admission-config
  mountPath: /etc/kubernetes/admission
```

Deploying an external policy engine

The Open Policy Agent (OPA) is an open source, general-purpose policy engine that you can use to enforce policy-based placement decisions in the Federation control plane.

Create a Service in the host cluster to contact the external policy engine:

```
kubectl create -f policy-engine-service.yaml
```

Shown below is an example Service for OPA.

policy-engine-service.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: opa
  namespace: federation-system
spec:
  selector:
    app: opa
  ports:
  - name: http
    protocol: TCP
    port: 8181
    targetPort: 8181
```

Create a Deployment in the host cluster with the Federation control plane:

```
kubectl create -f policy-engine-deployment.yaml
```

Shown below is an example Deployment for OPA.

policy-engine-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: opa
  name: opa
  namespace: federation-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: opa
      name: opa
    spec:
      containers:
        - name: opa
          image: openpolicyagent/opa:0.4.10
          args:
          - "run"
          - "--server"
        - name: kube-mgmt
          image: openpolicyagent/kube-mgmt:0.2
          args:
          - "-kubeconfig=/srv/kubernetes/kubeconfig"
          - "-cluster=federation/v1beta1/clusters"
          volumeMounts:
           - name: federation-kubeconfig
             mountPath: /srv/kubernetes
             readOnly: true
      volumes:
      - name: federation-kubeconfig
        secret:
          secretName: federation-controller-manager-kubeconfig
```

Configuring placement policies via ConfigMaps

The external policy engine will discover placement policies created in the `kube-federation-scheduling-policy` namespace in the Federation API server.

Create the namespace if it does not already exist:

```
kubectl --context=federation create namespace kube-federation-scheduling-policy
```

Configure a sample policy to test the external policy engine:

policy.rego

```
# OPA supports a high-level declarative language named Rego for authoring and
# enforcing policies. For more infomration on Rego, visit
# http://openpolicyagent.org.

# Rego policies are namespaced by the "package" directive.
package kubernetes.placement

# Imports provide aliases for data inside the policy engine. In this case, the
# policy simply refers to "clusters" below.
import data.kubernetes.clusters

# The "annotations" rule generates a JSON object containing the key
# "federation.kubernetes.io/replica-set-preferences" mapped to <preferences>.
# The preferences values is generated dynamically by OPA when it evaluates the
# rule.
#
# The SchedulingPolicy Admission Controller running inside the Federation API
# server will merge these annotatiosn into incoming Federated resources. By
# setting replica-set-preferences, we can control the placement of Federated
# ReplicaSets.
#
# Rules are defined to generate JSON values (booleans, strings, objects, etc.)
# When OPA evaluates a rule, it generates a value IF all of the expressions in
# the body evaluate successfully. All rules can be understood intuitively as
# <head> if <body> where <body> is true if <expr-1> AND <expr-2> AND ...
# <expr-N> is true (for some set of data.)
annotations["federation.kubernetes.io/replica-set-preferences"] = preferences {
    input.kind = "ReplicaSet"
    value = {"clusters": cluster_map, "rebalance": true}
    json.marshal(value, preferences)
}

# This "annotations" rule generates a value for the "federation.alpha.kubernetes.i
# annotation.
#
# In English, the policy asserts that resources in the "production" namespace
# that are not annotated with "criticality=low" MUST be placed on clusters
# labelled with "on-premises=true".
annotations["federation.alpha.kubernetes.io/cluster-selector"] = selector {
    input.metadata.namespace = "production"
    not input.metadata.annotations.criticality = "low"
    json.marshal([{
        "operator": "=",
        "key": "on-premises",
            "values": "[true]",
        }], selector)
}

# Generates a set of cluster names that satisfy the incoming Federated
# ReplicaSet's requirements. In this case, just PCI compliance.
replica_set_clusters[cluster_name] {
    clusters[cluster_name]
    not insufficient_pci[cluster_name]
}

# Generates a set of clusters that must not be used for Federated ReplicaSets
# that request PCI compliance.
insufficient_pci[cluster_name] {
    clusters[cluster_name]
    input.metadata.annotations["requires-pci"] = "true"
    not pci_clusters[cluster_name]
}

# Generates a set of clusters that are PCI certified. In this case, we assume
# clusters are annotated to indicate if they have passed PCI compliance audits.
pci_clusters[cluster_name] {
    clusters[cluster_name].metadata.annotations["pci-certified"] = "true"
}

# Helper rule to generate a mapping of desired clusters to weights. In this
# case, weights are static.
cluster_map[cluster_name] = {"weight": 1} {
    replica_set_clusters[cluster_name]
}
```

Shown below is the command to create the sample policy:

```
kubectl --context=federation -n kube-federation-scheduling-policy create configmap
```

This sample policy illustrates a few key ideas:

- Placement policies can refer to any field in Federated resources.
- Placement policies can leverage external context (for example, Cluster metadata) to make decisions.
- Administrative policy can be managed centrally.

- Policies can define simple interfaces (such as the `requires-pci` annotation) to avoid duplicating logic in manifests.

Testing placement policies

Annotate one of the clusters to indicate that it is PCI certified.

```
kubectl --context=federation annotate clusters cluster-name-1 pci-certified=true
```

Deploy a Federated ReplicaSet to test the placement policy.

replicaset-example-policy.yaml

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  labels:
    app: nginx-pci
  name: nginx-pci
  annotations:
    requires-pci: "true"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pci
  template:
    metadata:
      labels:
        app: nginx-pci
    spec:
      containers:
      - image: nginx
        name: nginx-pci
```

Shown below is the command to deploy a ReplicaSet that *does* match the policy.

```
kubectl --context=federation create -f replicaset-example-policy.yaml
```

Inspect the ReplicaSet to confirm the appropriate annotations have been applied:

```
kubectl --context=federation get rs nginx-pci -o jsonpath='{.metadata.annotations}
```

Federated Cluster

This guide explains how to use Clusters API resource in a Federation control plane.

Different than other Kubernetes resources, such as Deployments, Services and ConfigMaps, clusters only exist in the federation context, i.e. those requests must be submitted to the federation api-server.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general.

Listing Clusters

To list the clusters available in your federation, you can use kubectl by running:

```
kubectl --context=federation get clusters
```

The `--context=federation` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster. If you submit it to a k8s cluster, you will receive an error saying `the server doesn't have a resource type "clusters"`

If you passed the correct Federation context but received a message error saying `No resources found.`, it means that you haven't added any cluster to the Federation yet.

Creating a Federated Cluster

Creating a `cluster` resource in federation means joining it to the federation. To do so, you can use `kubefed join`. Basically, you need to give the new cluster a name and say what is the name of the context that corresponds to a cluster that hosts the federation. The following example command adds the cluster `gondor` to the federation running on host cluster `rivendell`:

```
kubefed join gondor --host-cluster-context=rivendell
```

You can find more details on how to do that in the respective section in the kubefed guide.

Deleting a Federated Cluster

Converse to creating a cluster, deleting a cluster means unjoining this cluster from the federation. This can be done with `kubefed unjoin` command. To remove the `gondor` cluster, just do:

```
kubefed unjoin gondor --host-cluster-context=rivendell
```

You can find more details on unjoin in the kubefed guide.

Labeling Clusters

You can label clusters the same way as any other Kubernetes object, which can help with grouping clusters and can also be leveraged by the ClusterSelector.

```
kubectl --context=rivendell label cluster gondor key1=value1 key2=value2
```

ClusterSelector Annotation

Starting in Kubernetes 1.7, there is alpha support for directing objects across the federated clusters with the annotation `federation.alpha.kubernetes.io/cluster-selector`. The *ClusterSelector* is conceptually similar to `nodeSelector`, but instead of selecting against labels on nodes, it selects against labels on federated clusters.

The annotation value must be JSON formatted and must be parsable into the ClusterSelector API type. For example: `[{"key": "load", "operator": "Lt", "values": ["10"]}]`. Content that doesn't parse correctly will throw an error and prevent distribution of the object to any federated clusters. Objects of type ConfigMap, Secret, Daemonset, Service and Ingress are included in the alpha implementation.

Here is an example ClusterSelector annotation, which will only select clusters WITH the label `pci=true` and WITHOUT the label `environment=test`:

```
metadata:
  annotations:
    federation.alpha.kubernetes.io/cluster-selector: '[{"key": "pci", "operator"
      "In", "values": ["true"]}, {"key": "environment", "operator": "NotIn", "va
      ["test"]}]'
```

The *key* is matched against label names on the federated clusters.

The *values* are matched against the label values on the federated clusters.

The possible *operators* are: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`.

The *values* field is expected to be empty when `Exists` or `DoesNotExist` is specified and may include more than one string when `In` or `NotIn` are used.

Currently, only integers are supported with `Gt` or `Lt`.

Clusters API reference

The full clusters API reference is currently in `federation/v1beta1` and more details can be found in the Federation API reference page.

Federated ConfigMap

This guide explains how to use ConfigMaps in a Federation control plane.

Federated ConfigMaps are very similar to the traditional Kubernetes ConfigMaps and provide the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general and ConfigMaps in particular.

Creating a Federated ConfigMap

The API for Federated ConfigMap is 100% compatible with the API for traditional Kubernetes ConfigMap. You can create a ConfigMap by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myconfigmap.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated ConfigMap is created, the federation control plane will create a matching ConfigMap in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get configmap myconfigmap
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone.

These ConfigMaps in underlying clusters will match the Federated ConfigMap.

Updating a Federated ConfigMap

You can update a Federated ConfigMap as you would update a Kubernetes ConfigMap; however, for a Federated ConfigMap, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated ConfigMap is updated, it updates the corresponding ConfigMaps in all underlying clusters to match it.

Deleting a Federated ConfigMap

You can delete a Federated ConfigMap as you would delete a Kubernetes ConfigMap; however, for a Federated ConfigMap, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete configmap
```

Note that at this point, deleting a Federated ConfigMap will not delete the corresponding ConfigMaps from underlying clusters. You must delete the underlying ConfigMaps manually. We intend to fix this in the future.

Federated DaemonSet

This guide explains how to use DaemonSets in a federation control plane.

DaemonSets in the federation control plane ("Federated Daemonsets" in this guide) are very similar to the traditional Kubernetes DaemonSets and provide the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and DaemonSets in particular.

Creating a Federated Daemonset

The API for Federated Daemonset is 100% compatible with the API for traditional Kubernetes DaemonSet. You can create a DaemonSet by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mydaemonset.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated Daemonset is created, the federation control plane will create a matching DaemonSet in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get daemonset mydaemonset
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone.

Updating a Federated Daemonset

You can update a Federated Daemonset as you would update a Kubernetes DaemonSet; however, for a Federated Daemonset, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated Daemonset is updated, it updates the corresponding DaemonSets in all underlying clusters to match it.

Deleting a Federated Daemonset

You can delete a Federated Daemonset as you would delete a Kubernetes DaemonSet; however, for a Federated Daemonset, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete daemonset mydaemonset
```

Federated Deployment

This guide explains how to use Deployments in the Federation control plane.

Deployments in the federation control plane (referred to as “Federated Deployments” in this guide) are very similar to the traditional Kubernetes Deployment and provide the same functionality. Creating them in the federation control plane ensures that the desired number of replicas exist across the registered clusters.

As of Kubernetes version 1.5, Federated Deployment is an Alpha feature. The core functionality of Deployment is present, but some features (such as full rollout compatibility) are still in development.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general and Deployments in particular.

Creating a Federated Deployment

The API for Federated Deployment is compatible with the API for traditional Kubernetes Deployment. You can create a Deployment by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mydeployment.yaml
```

The '–context=federation-cluster' flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated Deployment is created, the federation control plane will create a Deployment in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get deployment mydep
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone.

These Deployments in underlying clusters will match the federation Deployment *except* in the number of replicas and revision-related annotations. Federation control plane ensures that the sum of replicas in each cluster combined matches the desired number of replicas in the Federated Deployment.

Spreading Replicas in Underlying Clusters

By default, replicas are spread equally in all the underlying clusters. For example: if you have 3 registered clusters and you create a Federated Deployment with `spec.replicas = 9`, then each Deployment in the 3 clusters will have `spec.replicas=3`. To modify the number of replicas in each cluster, you can specify FederatedReplicaSetPreference as an annotation with key `federation.kubernetes.io/deployment-preferences` on Federated Deployment.

Updating a Federated Deployment

You can update a Federated Deployment as you would update a Kubernetes Deployment; however, for a Federated Deployment, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated Deployment is updated, it updates the corresponding Deployments in all underlying clusters to match it. So if the rolling update strategy was chosen then the underlying cluster will do the rolling update independently and `maxSurge` and `maxUnavailable` will apply only to individual clusters. This behavior may change in the future.

If your update includes a change in number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that their sum remains equal to the number of desired replicas in Federated Deployment.

Deleting a Federated Deployment

You can delete a Federated Deployment as you would delete a Kubernetes Deployment; however, for a Federated Deployment, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete deployment mydep
```

Federated Events

This guide explains how to use events in federation control plane to help in debugging.

Prerequisites

This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general.

Overview

Events in federation control plane (referred to as “federation events” in this guide) are very similar to the traditional Kubernetes Events providing the same functionality. Federation Events are stored only in federation control plane and are not passed on to the underlying Kubernetes clusters.

Federation controllers create events as they process API resources to surface to the user, the state that they are in. You can get all events from federation apiserver by running:

```
kubectl --context=federation-cluster get events
```

The standard kubectl get, update, delete commands will all work.

Federated Horizontal Pod Autoscalers (HPA)

FEATURE STATE: `Kubernetes v1.8` alpha

This guide explains how to use federated horizontal pod autoscalers (HPAs) in the federation control plane.

HPAs in the federation control plane are similar to the traditional Kubernetes HPAs, and provide the same functionality. Creating an HPA targeting a federated object in the federation control plane ensures that the desired number of replicas of the target object are scaled across the registered clusters, instead of a single cluster. Also, the control plane keeps monitoring the status of each individual HPA in the federated clusters and ensures the workload replicas move where they are needed most by manipulating the min and max limits of the HPA objects in the federated clusters.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and HPAs in particular.

The federated HPA is an alpha feature. The API is not enabled by default on the federated API server. To use this feature, the user or the admin deploying the federation control plane needs to run the federated API server with option `--runtime-config=api/all=true` to enable all APIs, including alpha APIs. Additionally, the federated HPA only works when used with CPU utilization metrics.

Creating a federated HPA

The API for federated HPAs is 100% compatible with the API for traditional Kubernetes HPA. You can create an HPA by sending a request to the federation API server.

You can do that with kubectl by running:

```
cat <<EOF | kubectl --context=federation-cluster create -f -
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
EOF
```

The `--context=federation-cluster` flag tells `kubectl` to submit the request to the federation API server instead of sending it to a Kubernetes cluster.

Once a federated HPA is created, the federation control plane partitions and creates the HPA in all underlying Kubernetes clusters. As of Kubernetes V1.7, cluster selectors can also be used to restrict any federated object, including the HPAs in a subset of clusters.

You can verify the creation by checking each of the underlying clusters. For example, with a context named `gce-asia-east1a` configured in your client for your cluster in that zone:

```
kubectl --context=gce-asia-east1a get HPA php-apache
```

The HPA in the underlying clusters will match the federation HPA except in the number of min and max replicas. The federation control plane ensures that the sum of max replicas in each cluster matches the specified max replicas on the federated HPA object, and the sum of minimum replicas will be greater than or equal to the minimum specified on the federated HPA object.

> **Note:** A particular cluster cannot have a minimum replica sum of 0.

Spreading HPA min and max replicas in underlying clusters

By default, first max replicas are spread equally in all the underlying clusters, then min replicas are distributed to those clusters that received their maximum value. This means that each cluster will get an HPA if the specified max replicas are greater than the total clusters participating in this federation, and some clusters will be skipped if specified max replicas are less than the total clusters participating in the federation.

For example: if you have 3 registered clusters and you create a federated HPA with `spec.maxReplicas = 9`, and `spec.minReplicas = 2`, then each HPA in the 3 clusters will get `spec.maxReplicas=3` and `spec.minReplicas = 1`.

Currently the default distribution is only available on the federated HPA, but in the future, users preferences could also be specified to control and/or restrict this distribution.

Updating a federated ReplicaSet

You can update a federated HPA as you would update a Kubernetes HPA; however, for a federated HPA, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. The Federation control plane ensures that whenever the federated HPA is updated, it updates the corresponding HPA in all underlying clusters to match it.

If your update includes a change in the number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that the sum of the max and min replicas remains matched as specified in the previous section.

Deleting a federated HPA

You can delete a federated HPA as you would delete a Kubernetes HPA; however, for a federated HPA, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. It should also be noted that for the federated resource to be deleted from all underlying clusters, cascading deletion should be used.

For example, you can do that using `kubectl` by running:

```
kubectl --context=federation-cluster delete HPA php-apache
```

Alternative ways to use federated HPA

To a federation user interacting with federated control plane (or simply federation), the interaction is almost identical to interacting with a normal Kubernetes cluster (but with a limited set of APIs that are federated). As both Deployments and HorizontalPodAutoscalers are now federated, `kubectl` commands like `kubectl run` and `kubectl autoscale` work on federation. Given this fact, the mechanism specified in horizontal pod autoscaler walkthrough will also work when used with federation. Care however will need to be taken that when generating load on a target deployment, it should be done against a specific federated cluster (or multiple clusters) not the federation.

Conclusion

The use of federated HPA is to ensure workload replicas move to the cluster(s) where they are needed most, or in other words where the load is beyond expected threshold. The federated HPA feature achieves this by manipulating the min and max replicas on the HPAs it creates in the federated clusters. It does not directly monitor the target object metrics from the federated clusters. It actually relies on the in-cluster HPA controllers to monitor the metrics and update relevant fields. The in-cluster HPA controller monitors the target pod metrics and updates the fields like desired replicas (after metrics based calculations) and current replicas (observing the current status of in cluster pods). The federated HPA controller, on the other hand, monitors only the cluster-specific HPA object fields and updates the min replica and max replica fields of those in cluster HPA objects, which have replicas matching thresholds.

For example, if a cluster has both desired replicas and current replicas the same as the max replicas, and averaged current CPU utilization still higher than the target CPU utilization (all of which are fields on local HPA object), then the target app in this cluster needs more replicas, and the scaling is currently restricted by max replicas set on this local HPA object. In such a scenario, the federated HPA controller scans all clusters and tries to find clusters which do not have such a condition (meaning the the desired replicas are less than the max, and current averaged cpu utilization is lower then the threshold). If it finds such a cluster, it reduces the max replica on the HPA in this cluster and increases the max replicas on the HPA in the cluster which needed the replicas.

There are many other similar conditions which the federated HPA controller checks and moves the max replicas and min replicas around the local HPAs in federated clusters to eventually ensure that the replicas move (or remain) in the cluster(s) which need them.

For more information, see "federated HPA design proposal".

Federated Ingress

This page explains how to use Kubernetes Federated Ingress to deploy a common HTTP(S) virtual IP load balancer across a federated service running in multiple Kubernetes clusters. As of v1.4, clusters hosted in Google Cloud (both GKE and GCE, or both) are supported. This makes it easy to deploy a service that reliably serves HTTP(S) traffic originating from web clients around the globe on a single, static IP address. Low network latency, high fault tolerance and easy administration are ensured through intelligent request routing and automatic replica relocation (using Federated ReplicaSets. Clients are automatically routed, via the shortest network path, to the cluster closest to them with available capacity (despite the fact that all clients use exactly the same static IP address). The load balancer automatically checks the health of the pods comprising the service, and avoids sending requests to unresponsive or slow pods (or entire unresponsive clusters).

Federated Ingress is released as an alpha feature, and supports Google Cloud Platform (GKE, GCE and hybrid scenarios involving both) in Kubernetes v1.4. Work is under way to support other cloud providers such as AWS, and other hybrid cloud scenarios (e.g. services spanning private on-premises as well as public cloud Kubernetes clusters).

You create Federated Ingresses in much that same way as traditional Kubernetes Ingresses: by making an API call which specifies the desired properties of your logical ingress point. In the case of Federated Ingress, this API call is directed to the Federation API endpoint, rather than a Kubernetes cluster API endpoint. The API for Federated Ingress is 100% compatible with the API for traditional Kubernetes Services.

Once created, the Federated Ingress automatically:

- Creates matching Kubernetes Ingress objects in every cluster underlying your Cluster Federation
- Ensures that all of these in-cluster ingress objects share the same logical global L7 (that is, HTTP(S)) load balancer and IP address
- Monitors the health and capacity of the service shards (that is, your pods) behind this ingress in each cluster
- Ensures that all client connections are routed to an appropriate healthy backend service endpoint at all times, even in the event of pod, cluster, availability zone or regional outages

Note that in the case of Google Cloud, the logical L7 load balancer is not a single physical device (which would present both a single point of failure, and a single global network routing choke point), but rather a truly global, highly available load balancing managed service, globally reachable via a single, static IP address.

Clients inside your federated Kubernetes clusters (Pods) will be automatically routed to the cluster-local shard of the Federated Service backing the Ingress in their cluster if it exists and is healthy, or the closest healthy shard in a different cluster if it does not. Note that this involves a network trip to the HTTP(s) load balancer, which resides outside your local Kubernetes cluster but inside the same GCP region.

Before you begin

This document assumes that you have a running Kubernetes Cluster Federation installation. If not, then see the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You must also have a basic working knowledge of Kubernetes in general, and Ingress in particular.

Creating a federated ingress

You can create a federated ingress in any of the usual ways, for example, using kubectl:

```
kubectl --context=federation-cluster create -f myingress.yaml
```

For example ingress YAML configurations, see the Ingress User Guide. The '–context=federation-cluster' flag tells kubectl to submit the request to the Federation API endpoint, with the appropriate credentials. If you have not yet configured such a context, see the federation admin guide or one of the administration tutorials to find out how to do so.

The Federated Ingress automatically creates and maintains matching Kubernetes ingresses in all of the clusters underlying your federation. These cluster-specific ingresses (and their associated ingress controllers) configure and manage the load balancing and health checking infrastructure that ensures that traffic is load balanced to each cluster appropriately.

You can verify this by checking in each of the underlying clusters. For example:

```
kubectl --context=gce-asia-east1a get ingress myingress
NAME        HOSTS     ADDRESS           PORTS     AGE
myingress   *         130.211.5.194     80, 443   1m
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone. The name and namespace of the underlying ingress automatically matches those of the Federated Ingress that you created above (and if you happen to have had ingresses of the same name and namespace already existing in any of those clusters, they will be automatically adopted by the Federation and updated to conform with the specification of your Federated Ingress. Either way, the end result will be the same).

The status of your Federated Ingress automatically reflects the real-time status of the underlying Kubernetes ingresses. For example:

```
kubectl --context=federation-cluster describe ingress myingress

Name:           myingress
Namespace:      default
Address:        130.211.5.194
TLS:
  tls-secret terminates
Rules:
  Host  Path    Backends
  ----  ----    --------
  * *   echoheaders-https:80 (10.152.1.3:8080,10.152.2.4:8080)
Annotations:
  https-target-proxy:       k8s-tps-default-myingress--ff1107f83ed600c0
  target-proxy:         k8s-tp-default-myingress--ff1107f83ed600c0
  url-map:          k8s-um-default-myingress--ff1107f83ed600c0
  backends:         {"k8s-be-30301--ff1107f83ed600c0":"Unknown"}
  forwarding-rule:      k8s-fw-default-myingress--ff1107f83ed600c0
  https-forwarding-rule:    k8s-fws-default-myingress--ff1107f83ed600c0
Events:
  FirstSeen LastSeen    Count   From                SubobjectPath   Type        Re
  --------- --------    -----   ----                -------------   ---------   --
  3m        3m      1   {loadbalancer-controller }          Normal      ADD defaul
  2m        2m      1   {loadbalancer-controller }          Normal      CREATE   ip
```

Note that:

- The address of your Federated Ingress corresponds with the address of all of the underlying Kubernetes ingresses (once these have been allocated - this may take up to a few minutes).
- You have not yet provisioned any backend Pods to receive the network traffic directed to this ingress (that is, ‘Service Endpoints’ behind the service backing the Ingress), so the Federated Ingress does not yet consider these to be healthy shards and will not direct traffic to any of these clusters.
- The federation control system automatically reconfigures the load balancer controllers in all of the clusters in your federation to make them consistent, and allows them to share global load balancers. But this reconfiguration can only complete successfully if there are no pre-existing Ingresses in those clusters (this is a safety feature to prevent accidental breakage of existing ingresses). So, to ensure that your federated ingresses function correctly, either start with new, empty clusters, or make sure that you delete (and recreate if necessary) all pre-existing Ingresses in the clusters comprising your federation.

Adding backend services and pods

To render the underlying ingress shards healthy, you need to add backend Pods behind the service upon which the Ingress is based. There are several ways to achieve this, but the easiest is to create a Federated Service and Federated ReplicaSet. To create appropriately labelled pods and services in the 13 underlying clusters of your federation:

```
kubectl --context=federation-cluster create -f services/nginx.yaml
```

```
kubectl --context=federation-cluster create -f myreplicaset.yaml
```

Note that in order for your federated ingress to work correctly on Google Cloud, the node ports of all of the underlying cluster-local services need to be identical. If you're using a federated service this is easy to do. Simply pick a node port that is not already being used in any of your clusters, and add that to the spec of your federated service. If you do not specify a node port for your federated service, each cluster will choose its own node port for its cluster-local shard of the service, and these will probably end up being different, which is not what you want.

You can verify this by checking in each of the underlying clusters. For example:

```
kubectl --context=gce-asia-east1a get services nginx
NAME      CLUSTER-IP     EXTERNAL-IP      PORT(S)   AGE
nginx     10.63.250.98   104.199.136.89   80/TCP    9m
```

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (for example, Google Cloud, AWS), and on-premises (for example, on OpenStack). However, in Kubernetes v1.4, Federated Ingress is only supported across Google Cloud clusters.

Discovering a federated ingress

Ingress objects (in both plain Kubernets clusters, and in federations of clusters) expose one or more IP addresses (via the Status.Loadbalancer.Ingress field) that remains static for the lifetime of the Ingress object (in future, automatically managed DNS names might also be added). All clients (whether internal to your cluster, or on the external network or internet) should connect to one of these IP or DNS addresses. All client requests are automatically routed, via the shortest network path, to a healthy pod in the closest cluster to the origin of the request. So for example, HTTP(S) requests from internet users in Europe will be routed directly to the closest cluster in Europe that has available capacity. If there are no such clusters in Europe, the request will be routed to the next closest cluster (typically in the U.S.).

Handling failures of backend pods and whole clusters

Ingresses are backed by Services, which are typically (but not always) backed by one or more ReplicaSets. For Federated Ingresses, it is common practise to use the federated variants of Services and ReplicaSets for this purpose.

In particular, Federated ReplicaSets ensure that the desired number of pods are kept running in each cluster, even in the event of node failures. In the event of entire cluster or availability zone failures, Federated ReplicaSets automatically place additional replicas in the other available clusters in the federation to accommodate the traffic which was previously being served by the now unavailable cluster. While the Federated ReplicaSet ensures that sufficient replicas are kept running, the Federated Ingress ensures that user traffic is automatically redirected away from the failed cluster to other available clusters.

Troubleshooting

I cannot connect to my cluster federation API.

Check that your:

1. Client (typically `kubectl`) is correctly configured (including API endpoints and login credentials).
2. Cluster Federation API server is running and network-reachable.

See the federation admin guide to learn how to bring up a cluster federation correctly (or have your cluster administrator do this for you), and how to correctly configure your client.

I can create a Federated Ingress/service/replicaset successfully against the cluster federation API, but no matching ingresses/services/replicasets are created in my underlying clusters.

Check that:

1. Your clusters are correctly registered in the Cluster Federation API. (`kubectl describe clusters`)
2. Your clusters are all 'Active'. This means that the cluster Federation system was able to connect and authenticate against the clusters' endpoints. If not, consult the event logs of the federation-controller-manager pod to ascertain what the failure might be. (`kubectl --namespace=federation logs $(kubectl get pods --namespace=federation -l module=federation-controller-manager -o name`)
3. That the login credentials provided to the Cluster Federation API for the clusters have the correct authorization and quota to create ingresses/services/replicasets in the relevant namespace in the clusters. Again you should see associated error messages providing more detail in the above event log file if this is not the case.
4. Whether any other error is preventing the service creation operation from succeeding (look for `ingress-controller` , `service-controller` or `replicaset-controller` , errors in the output of `kubectl logs federation-controller-manager --namespace federation`).

I can create a federated ingress successfully, but request load is not correctly distributed across the underlying clusters.

Check that:

1. The services underlying your federated ingress in each cluster have identical node ports. See above for further explanation.
2. The load balancer controllers in each of your clusters are of the correct type ("GLBC") and have been correctly reconfigured by the federation control plane to share a global GCE load balancer (this should happen automatically). If they are of the correct type, and have been correctly reconfigured, the UID data item in the GLBC configmap in each cluster will be identical across all

clusters. See the GLBC docs for further details. If this is not the case, check the logs of your federation controller manager to determine why this automated reconfiguration might be failing.

3. No ingresses have been manually created in any of your clusters before the above reconfiguration of the load balancer controller completed successfully. Ingresses created before the reconfiguration of your GLBC will interfere with the behavior of your federated ingresses created after the reconfiguration (see the GLBC docs for further information). To remedy this, delete any ingresses created before the cluster joined the federation (and had its GLBC reconfigured), and recreate them if necessary.

What's next

- If you need assistance, use one of the support channels to seek assistance.
- For details about use cases that motivated this work, see Federation proposal.

Federated Jobs

This guide explains how to use jobs in the federation control plane.

Jobs in the federation control plane (referred to as “federated jobs” in this guide) are similar to the traditional Kubernetes jobs, and provide the same functionality. Creating jobs in the federation control plane ensures that the desired number of parallelism and completions exist across the registered clusters.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and jobs in particular.

Creating a federated job

The API for federated jobs is fully compatible with the API for traditional Kubernetes jobs. You can create a job by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myjob.yaml
```

The '–context=federation-cluster' flag tells kubectl to submit the request to the federation API server instead of sending it to a Kubernetes cluster.

Once a federated job is created, the federation control plane creates a job in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get job myjob
```

The previous example assumes that you have a context named `gce-asia-east1a` configured in your client for your cluster in that zone.

The jobs in the underlying clusters match the federated job except in the number of parallelism and completions. The federation control plane ensures that the sum of the parallelism and completions in each cluster matches the desired number of parallelism and completions in the federated job.

Spreading job tasks in underlying clusters

By default, parallelism and completions are spread equally in all underlying clusters. For example: if you have 3 registered clusters and you create a federated job with `spec.parallelism = 9` and `spec.completions = 18`, then each job in the 3 clusters has `spec.parallelism = 3` and `spec.completions = 6`. To modify the number of parallelism and completions in each cluster, you can specify ReplicaAllocationPreferences as an annotation with key `federation.kubernetes.io/job-preferences` on the federated job.

Updating a federated job

You can update a federated job as you would update a Kubernetes job; however, for a federated job, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the federated job is updated, it updates the corresponding job in all underlying clusters to match it.

If your update includes a change in number of parallelism and completions, the federation control plane changes the number of parallelism and completions in underlying clusters to ensure that their sum remains equal to the number of desired parallelism and completions in federated job.

Deleting a federated job

You can delete a federated job as you would delete a Kubernetes job; however, for a federated job, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster.

For example, with kubectl:

```
kubectl --context=federation-cluster delete job myjob
```

Note: Deleting a federated job will not delete the corresponding jobs from underlying clusters. You must delete the underlying jobs manually.

Federated Namespaces

This guide explains how to use Namespaces in Federation control plane.

Namespaces in federation control plane (referred to as "federated Namespaces" in this guide) are very similar to the traditional Kubernetes Namespaces providing the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and Namespaces in particular.

Creating a Federated Namespace

The API for Federated Namespaces is 100% compatible with the API for traditional Kubernetes Namespaces. You can create a Namespace by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myns.yaml
```

The '–context=federation-cluster' flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated Namespace is created, the federation control plane will create a matching Namespace in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get namespaces myns
```

The above assumes that you have a context named 'gce-asia-east1a' configured in your client for your cluster in that zone. The name and spec of the underlying Namespace will match those of the Federated Namespace that you created above.

Updating a Federated Namespace

You can update a federated Namespace as you would update a Kubernetes Namespace, just send the request to federation apiserver instead of sending it to a specific Kubernetes cluster. Federation control plan will ensure that whenever the federated Namespace is updated, it updates the corresponding Namespaces in all underlying clusters to match it.

Deleting a Federated Namespace

You can delete a federated Namespace as you would delete a Kubernetes Namespace, just send the request to federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete ns myns
```

As in Kubernetes, deleting a federated Namespace will delete all resources in that Namespace from the federation control plane.

Note that at this point, deleting a federated Namespace will not delete the corresponding Namespace and resources in those Namespaces from underlying clusters. Users are expected to delete them manually. We intend to fix this in the future.

Federated ReplicaSets

This guide explains how to use ReplicaSets in the Federation control plane.

ReplicaSets in the federation control plane (referred to as “federated ReplicaSets” in this guide) are very similar to the traditional Kubernetes ReplicaSets, and provide the same functionality. Creating them in the federation control plane ensures that the desired number of replicas exist across the registered clusters.

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and ReplicaSets in particular.

Creating a Federated ReplicaSet

The API for Federated ReplicaSet is 100% compatible with the API for traditional Kubernetes ReplicaSet. You can create a ReplicaSet by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myrs.yaml
```

The ‘–context=federation-cluster’ flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated ReplicaSet is created, the federation control plane will create a ReplicaSet in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get rs myrs
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

The ReplicaSets in the underlying clusters will match the federation ReplicaSet except in the number of replicas. The federation control plane will ensure that the sum of the replicas in each cluster match the desired number of replicas in the federation ReplicaSet.

Spreading Replicas in Underlying Clusters

By default, replicas are spread equally in all the underlying clusters. For example: if you have 3 registered clusters and you create a federated ReplicaSet with `spec.replicas = 9`, then each ReplicaSet in the 3 clusters will have `spec.replicas=3`. To modify the number of replicas in each cluster, you can specify FederatedReplicaSetPreference as an annotation with key `federation.kubernetes.io/replica-set-preferences` on the federated ReplicaSet.

Updating a Federated ReplicaSet

You can update a federated ReplicaSet as you would update a Kubernetes ReplicaSet; however, for a federated ReplicaSet, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The Federation control plane ensures that whenever the federated ReplicaSet is updated, it updates the corresponding ReplicaSet in all underlying clusters to match it. If your update includes a change in number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that their sum remains equal to the number of desired replicas in federated ReplicaSet.

Deleting a Federated ReplicaSet

You can delete a federated ReplicaSet as you would delete a Kubernetes ReplicaSet; however, for a federated ReplicaSet, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete rs myrs
```

Note that at this point, deleting a federated ReplicaSet will not delete the corresponding ReplicaSets from underlying clusters. You must delete the underlying ReplicaSets manually. We intend to fix this in the future.

Federated Secrets

This guide explains how to use secrets in Federation control plane.

Prerequisites

This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general and Secrets in particular.

Overview

Secrets in federation control plane (referred to as "federated secrets" in this guide) are very similar to the traditional Kubernetes Secrets providing the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

Creating a Federated Secret

The API for Federated Secret is 100% compatible with the API for traditional Kubernetes Secret. You can create a secret by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mysecret.yaml
```

The ‘–context=federation-cluster’ flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated secret is created, the federation control plane will create a matching secret in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get secret mysecret
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

These secrets in underlying clusters will match the federated secret.

Updating a Federated Secret

You can update a federated secret as you would update a Kubernetes secret; however, for a federated secret, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The Federation control plan ensures that whenever the federated secret is updated, it updates the corresponding secrets in all underlying clusters to match it.

Deleting a Federated Secret

You can delete a federated secret as you would delete a Kubernetes secret; however, for a federated secret, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete secret mysecret
```

Note that at this point, deleting a federated secret will not delete the corresponding secrets from underlying clusters. You must delete the underlying secrets manually. We intend to fix this in the future.

Perform a Rolling Update on a DaemonSet

This page shows how to perform a rolling update on a DaemonSet.

Before you begin

- The DaemonSet rolling update feature is only supported in Kubernetes version 1.6 or later.

DaemonSet Update Strategy

DaemonSet has two update strategy types :

- OnDelete: This is the default update strategy for backward-compatibility. With **`OnDelete`** update strategy, after you update a DaemonSet template, new DaemonSet pods will *only* be created when you manually delete old DaemonSet pods. This is the same behavior of DaemonSet in Kubernetes version 1.5 or before.

- RollingUpdate: With `RollingUpdate` update strategy, after you update a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically, in a controlled fashion.

Caveat: Updating DaemonSet created from Kubernetes version 1.5 or before

If you try a rolling update on a DaemonSet that was created from Kubernetes version 1.5 or before, a rollout will be triggered when you *first* change the DaemonSet update strategy to `RollingUpdate`, no matter if DaemonSet template is modified or not. If the DaemonSet template is not changed, all existing DaemonSet pods will be restarted (deleted and created).

Therefore, make sure you want to trigger a rollout before you first switch the strategy to `RollingUpdate`.

Performing a Rolling Update

To enable the rolling update feature of a DaemonSet, you must set its `.spec.updateStrategy.type` to `RollingUpdate`.

You may want to set `.spec.updateStrategy.rollingUpdate.maxUnavailable` (default to 1) and `.spec.minReadySeconds` (default to 0) as well.

Step 1: Checking DaemonSet **RollingUpdate** update strategy

First, check the update strategy of your DaemonSet, and make sure it's set to RollingUpdate:

```
kubectl get ds/<daemonset-name> -o go-template='{{.spec.updateStrategy.type}}{{"\n
```

If you haven't created the DaemonSet in the system, check your DaemonSet manifest with the following command instead:

```
kubectl create -f ds.yaml --dry-run -o go-template='{{.spec.updateStrategy.type}}{
```

The output from both commands should be:

```
RollingUpdate
```

If the output isn't `RollingUpdate`, go back and modify the DaemonSet object or manifest accordingly.

Step 2: Creating a DaemonSet with **RollingUpdate** update strategy

If you have already created the DaemonSet, you may skip this step and jump to step 3.

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet:

```
kubectl create -f ds.yaml
```

Alternatively, use `kubectl apply` to create the same DaemonSet if you plan to update the DaemonSet with `kubectl apply`.

```
kubectl apply -f ds.yaml
```

Step 3: Updating a DaemonSet template

Any updates to a `RollingUpdate` DaemonSet `.spec.template` will trigger a rolling update. This can be done with several different `kubectl` commands.

Declarative commands

If you update DaemonSets using configuration files, use `kubectl apply`:

```
kubectl apply -f ds-v2.yaml
```

Imperative commands

If you update DaemonSets using imperative commands, use `kubectl edit` or `kubectl patch`:

```
kubectl edit ds/<daemonset-name>
```

```
kubectl patch ds/<daemonset-name> -p=<strategic-merge-patch>
```

Updating only the container image

If you just need to update the container image in the DaemonSet template, i.e. `.spec.template.spec.containers[*].image`, use `kubectl set image`:

```
kubectl set image ds/<daemonset-name> <container-name>=<container-new-image>
```

Step 4: Watching the rolling update status

Finally, watch the rollout status of the latest DaemonSet rolling update:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollout is complete, the output is similar to this:

```
daemonset "<daemonset-name>" successfully rolled out
```

Troubleshooting

DaemonSet rolling update is stuck

Sometimes, a DaemonSet rolling update may be stuck. Here are some possible causes:

Some nodes run out of resources

The rollout is stuck because new DaemonSet pods can't be scheduled on at least one node. This is possible when the node is running out of resources.

When this happens, find the nodes that don't have the DaemonSet pods scheduled on by comparing the output of `kubectl get nodes` and the output of:

```
kubectl get pods -l <daemonset-selector-key>=<daemonset-selector-value> -o wide
```

Once you've found those nodes, delete some non-DaemonSet pods from the node to make room for new DaemonSet pods. Note that this will cause service disruption if the deleted pods are not controlled by any controllers, or if the pods aren't replicated. This doesn't respect PodDisruptionBudget either.

Broken rollout

If the recent DaemonSet template update is broken, for example, the container is crash looping, or the container image doesn't exist (often due to a typo), DaemonSet rollout won't progress.

To fix this, just update the DaemonSet template again. New rollout won't be blocked by previous unhealthy rollouts.

Clock skew

If `.spec.minReadySeconds` is specified in the DaemonSet, clock skew between master and nodes will make DaemonSet unable to detect the right rollout progress.

What's next

- See Task: Performing a rollback on a DaemonSet
- See Concepts: Creating a DaemonSet to adopt existing DaemonSet pods

Performing a Rollback on a DaemonSet

This page shows how to perform a rollback on a DaemonSet.

Before you begin

- The DaemonSet rollout history and DaemonSet rollback features are only supported in `kubectl` in Kubernetes version 1.7 or later.
- Make sure you know how to perform a rolling update on a DaemonSet.

Performing a Rollback on a DaemonSet

Step 1: Find the DaemonSet revision you want to roll back to

You can skip this step if you just want to roll back to the last revision.

List all revisions of a DaemonSet:

```
kubectl rollout history daemonset <daemonset-name>
```

This returns a list of DaemonSet revisions:

```
daemonsets "<daemonset-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

- Change cause is copied from DaemonSet annotation `kubernetes.io/change-cause` to its revisions upon creation. You may specify `--record=true` in `kubectl` to record the command executed in the change cause annotation.

To see the details of a specific revision:

```
kubectl rollout history daemonset <daemonset-name> --revision=1
```

This returns the details of that revision:

```
daemonsets "<daemonset-name>" with revision #1
Pod Template:
Labels:       foo=bar
Containers:
app:
 Image:       ...
 Port:        ...
 Environment: ...
 Mounts:      ...
Volumes:       ...
```

Step 2: Roll back to a specific revision

```
# Specify the revision number you get from Step 1 in --to-revision
kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>
```

If it succeeds, the command returns:

```
daemonset "<daemonset-name>" rolled back
```

If `--to-revision` flag is not specified, the last revision will be picked.

Step 3: Watch the progress of the DaemonSet rollback

`kubectl rollout undo daemonset` tells the server to start rolling back the DaemonSet. The real rollback is done asynchronously on the server side.

To watch the progress of the rollback:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollback is complete, the output is similar to this:

```
daemonset "<daemonset-name>" successfully rolled out
```

Understanding DaemonSet Revisions

In the previous `kubectl rollout history` step, you got a list of DaemonSet revisions. Each revision is stored in a resource named `ControllerRevision`. `ControllerRevision` is a resource only available in Kubernetes release 1.7 or later.

To see what is stored in each revision, find the DaemonSet revision raw resources:

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-selector-val
```

This returns a list of `ControllerRevisions`:

```
NAME                               CONTROLLER                     REVISION   AGE
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-name>     1          1h
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-name>     2          1h
```

Each `ControllerRevision` stores the annotations and template of a DaemonSet revision.

`kubectl rollout undo` takes a specific `ControllerRevision` and replaces DaemonSet template with the template stored in the `ControllerRevision`. `kubectl rollout undo` is equivalent to updating DaemonSet template to a previous revision through other commands, such as `kubectl edit` or `kubectl apply`.

Note that DaemonSet revisions only roll forward. That is to say, after a rollback is complete, the revision number (`.revision` field) of the `ControllerRevision` being rolled back to will advance. For example, if you have revision 1 and 2 in the system, and roll back from revision 2 to revision 1, the `ControllerRevision` with `.revision: 1` will become `.revision: 3`.

Troubleshooting

- See troubleshooting DaemonSet rolling update.

Schedule GPUs

Kubernetes includes **experimental** support for managing NVIDIA GPUs spread across nodes. This page describes how users can consume GPUs and the current limitations.

Before you begin

1. Kubernetes nodes have to be pre-installed with Nvidia drivers. Kubelet will not detect Nvidia GPUs otherwise. Try to re-install Nvidia drivers if kubelet fails to expose Nvidia GPUs as part of Node Capacity. After installing the driver, run `nvidia-docker-plugin` to confirm that all drivers have been loaded.
2. A special **alpha** feature gate `Accelerators` has to be set to true across the system: `--feature-gates="Accelerators=true"` .
3. Nodes must be using `docker engine` as the container runtime.

The nodes will automatically discover and expose all Nvidia GPUs as a schedulable resource.

API

Nvidia GPUs can be consumed via container level resource requirements using the resource name `alpha.kubernetes.io/nvidia-gpu` .

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
    -
      name: gpu-container-1
      image: gcr.io/google_containers/pause:2.0
      resources:
        limits:
          alpha.kubernetes.io/nvidia-gpu: 2 # requesting 2 GPUs
    -
      name: gpu-container-2
      image: gcr.io/google_containers/pause:2.0
      resources:
        limits:
          alpha.kubernetes.io/nvidia-gpu: 3 # requesting 3 GPUs
```

- GPUs can be specified in the `limits` section only.
- Containers (and pods) do not share GPUs.
- Each container can request one or more GPUs.
- It is not possible to request a portion of a GPU.
- Nodes are expected to be homogenous, i.e. run the same GPU hardware.

If your nodes are running different versions of GPUs, then use Node Labels and Node Selectors to schedule pods to appropriate GPUs. Following is an illustration of this workflow:

As part of your Node bootstrapping, identify the GPU hardware type on your nodes and expose it as a node label.

```
NVIDIA_GPU_NAME=$(nvidia-smi --query-gpu=gpu_name --format=csv,noheader --id=0 | s
source /etc/default/kubelet
KUBELET_OPTS="$KUBELET_OPTS --node-labels='alpha.kubernetes.io/nvidia-gpu-name=$NV
echo "KUBELET_OPTS=$KUBELET_OPTS" > /etc/default/kubelet
```

Specify the GPU types a pod can use via [Node Affinity]() rules.

```
kind: pod
apiVersion: v1
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/affinity: >
      {
        "nodeAffinity": {
          "requiredDuringSchedulingIgnoredDuringExecution": {
            "nodeSelectorTerms": [
              {
                "matchExpressions": [
                  {
                    "key": "alpha.kubernetes.io/nvidia-gpu-name",
                    "operator": "In",
                    "values": ["Tesla K80", "Tesla P100"]
                  }
                ]
              }
            ]
          }
        }
      }
spec:
  containers:
    -
      name: gpu-container-1
      resources:
        limits:
          alpha.kubernetes.io/nvidia-gpu: 2
```

This will ensure that the pod will be scheduled to a node that has a `Tesla K80` or a `Tesla P100` Nvidia GPU.

Warning

The API presented here **will change** in an upcoming release to better support GPUs, and hardware accelerators in general, in Kubernetes.

Access to CUDA libraries

As of now, CUDA libraries are expected to be pre-installed on the nodes.

To mitigate this, you can copy the libraries to a more permissive folder in `/var/lib/` or change the permissions directly. (Future releases will automatically perform this operation)

Pods can access the libraries using `hostPath` volumes.

```
kind: Pod
apiVersion: v1
metadata:
  name: gpu-pod
spec:
  containers:
  - name: gpu-container-1
    image: gcr.io/google_containers/pause:2.0
    resources:
      limits:
        alpha.kubernetes.io/nvidia-gpu: 1
    volumeMounts:
    - mountPath: /usr/local/nvidia/bin
      name: bin
    - mountPath: /usr/lib/nvidia
      name: lib
  volumes:
  - hostPath:
      path: /usr/lib/nvidia-375/bin
    name: bin
  - hostPath:
      path: /usr/lib/nvidia-375
    name: lib
```

Future

- Support for hardware accelerators is in its early stages in Kubernetes.
- GPUs and other accelerators will soon be a native compute resource across the system.
- Better APIs will be introduced to provision and consume accelerators in a scalable manner.
- Kubernetes will automatically ensure that applications consuming GPUs gets the best possible performance.
- Key usability problems like access to CUDA libraries will be addressed.

Manage HugePages

FEATURE STATE: `Kubernetes v1.8` alpha

Kubernetes supports the allocation and consumption of pre-allocated huge pages by applications in a Pod as an **alpha** feature. This page describes how users can consume huge pages and the current limitations.

Before you begin

1. Kubernetes nodes must pre-allocate huge pages in order for the node to report its huge page capacity. A node may only pre-allocate huge pages for a single size.
2. A special **alpha** feature gate `HugePages` has to be set to true across the system: `--feature-gates="HugePages=true"`.

The nodes will automatically discover and report all huge page resources as a schedulable resource.

API

Huge pages can be consumed via container level resource requirements using the resource name `hugepages-<size>`, where size is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB page sizes, it will expose a schedulable resource `hugepages-2Mi`. Unlike CPU or memory, huge pages do not support overcommit.

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - image: fedora:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 100Mi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

- Huge page requests must equal the limits. This is the default if limits are specified, but requests are not.
- Huge pages are isolated at a pod scope, container isolation is planned in a future iteration.
- EmptyDir volumes backed by huge pages may not consume more huge page memory than the pod request.
- Applications that consume huge pages via `shmget()` with `SHM_HUGETLB` must run with a supplemental group that matches `proc/sys/vm/hugetlb_shm_group`

Future

- Support container isolation of huge pages in addition to pod isolation.
- NUMA locality guarnatees as a feature of quality of service.
- ResourceQuota support.
- LimitRange support.

Extend kubectl with plugins

FEATURE STATE: `Kubernetes v1.8` alpha

This guide shows you how to install and write extensions for kubectl. Usually called *plugins* or *binary extensions*, this feature allows you to extend the default set of commands available in `kubectl` by adding new subcommands to perform new tasks and extend the set of features available in the main distribution of `kubectl`.

Before you begin

You need to have a working `kubectl` binary installed. Note that plugins were officially introduced as an alpha feature in the v1.8.0 release. So, while some parts of the plugins feature were already available in previous versions, a `kubectl` version of 1.8.0 or later is recommended.

Until a GA version is released, plugins will only be available under the `kubectl plugin` subcommand.

Installing kubectl plugins

A plugin is nothing more than a set of files: at least a **plugin.yaml** descriptor, and likely one or more binary, script, or assets files. To install a plugin, copy those files to one of the locations in the filesystem where `kubectl` searches for plugins.

Note that Kubernetes does not provide a package manager or something similar to install or update plugins, so it's your responsibility to place the plugin files in the correct location. We recommend that each plugin is located on its own directory, so installing a plugin that is distributed as a compressed file is as simple as extracting it to one of the locations specified in the Plugin loader section.

Plugin loader

The plugin loader is responsible for searching plugin files in the filesystem locations specified below, and checking if the plugin provides the minimum amount of information required for it to run. Files placed in the right location that don't provide the minimum amount of information, for example an incomplete *plugin.yaml* descriptor, are ignored.

Search order

The plugin loader uses the following search order:

1. `${KUBECTL_PLUGINS_PATH}` If specified, the search stops here.
2. `${XDG_DATA_DIRS}/kubectl/plugins`
3. `~/.kube/plugins`

If the `KUBECTL_PLUGINS_PATH` environment variable is present, the loader uses it as the only location to look for plugins. The `KUBECTL_PLUGINS_PATH` environment variable is a list of directories. In Linux and Mac, the list is colon-delimited. In Windows, the list is semicolon-delimited.

If `KUBECTL_PLUGINS_PATH` is not present, the loader searches these additional locations:

First, one or more directories specified according to the XDG System Directory Structure specification. Specifically, the loader locates the directories specified by the `XDG_DATA_DIRS` environment variable, and then searches `kubectl/plugins` directory inside of those. If `XDG_DATA_DIRS` is not specified, it defaults to `/usr/local/share:/usr/share`.

Second, the `plugins` directory under the user's kubeconfig dir. In most cases, this is `~/.kube/plugins`.

```
# Loads plugins from both /path/to/dir1 and /path/to/dir2
KUBECTL_PLUGINS_PATH=/path/to/dir1:/path/to/dir2 kubectl plugin -h
```

Writing kubectl plugins

You can write a plugin in any programming language or script that allows you to write command-line commands. A plugin does not necessarily need to have a binary component. It could rely entirely on operating system utilities like `echo`, `sed`, or `grep`. Or it could rely on the `kubectl` binary.

The only strong requirement for a `kubectl` plugin is the `plugin.yaml` descriptor file. This file is responsible for declaring at least the minimum attributes required to register a plugin and must be located under one of the locations specified in the Search order section.

The plugin.yaml descriptor

The descriptor file supports the following attributes:

```
name: "targaryen"                 # REQUIRED: the plugin command name, to be invok
shortDesc: "Dragonized plugin"    # REQUIRED: the command short description, for h
longDesc: ""                      # the command long description, for help
example: ""                       # command example(s), for help
command: "./dracarys"             # REQUIRED: the command, binary, or script to in
flags:                            # flags supported by the plugin
  - name: "heat"                  # REQUIRED for each flag: flag name
    shorthand: "h"                # short version of the flag name
    desc: "Fire heat"             # REQUIRED for each flag: flag description
    defValue: "extreme"           # default value of the flag
tree:                             # allows the declaration of subcommands
  - ...                           # subcommands support the same set of attributes
```

The preceding descriptor declares the `kubectl plugin targaryen` plugin, which has one flag named `-h | --heat`. When the plugin is invoked, it calls the `dracarys` binary or script, which is located in the same directory as the descriptor file. The Accessing runtime attributes section describes how the `dracarys` command accesses the flag value and other runtime context.

Recommended directory structure

It is recommended that each plugin has its own subdirectory in the filesystem, preferably with the same name as the plugin command. The directory must contain the `plugin.yaml` descriptor and any binary, script, asset, or other dependency it might require.

For example, the directory structure for the `targaryen` plugin could look like this:

```
~/.kube/plugins/
└── targaryen
    ├── plugin.yaml
    └── dracarys
```

Accessing runtime attributes

In most use cases, the binary or script file you write to support the plugin must have access to some contextual information provided by the plugin framework. For example, if you declared flags in the descriptor file, your plugin must have access to the user-provided flag values at runtime. The same is true for global flags. The plugin framework is responsible for doing that, so plugin writers don't need to worry about parsing arguments. This also ensures the best level of consistency between plugins and regular `kubectl` commands.

Plugins have access to runtime context attributes through environment variables. So to access the value provided through a flag, for example, just look for the value of the proper environment variable using the appropriate function call for your binary or script.

The supported environment variables are:

- `KUBECTL_PLUGINS_CALLER`: The full path to the `kubectl` binary that was used in the current command invocation. As a plugin writer, you don't have to implement logic to authenticate and access the Kubernetes API. Instead, you can invoke `kubectl` to obtain the information you need, through something like `kubectl get --raw=/apis`.
- `KUBECTL_PLUGINS_CURRENT_NAMESPACE`: The current namespace that is the context for this call. This is the actual namespace to be used, meaning it was already processed in terms of the precedence between what was provided through the kubeconfig, the `--namespace` global flag, environment variables, and so on.
- `KUBECTL_PLUGINS_DESCRIPTOR_*`: One environment variable for every attribute declared in the `plugin.yaml` descriptor. For example, `KUBECTL_PLUGINS_DESCRIPTOR_NAME`, `KUBECTL_PLUGINS_DESCRIPTOR_COMMAND`.

- `KUBECTL_PLUGINS_GLOBAL_FLAG_*` : One environment variable for every global flag supported by `kubectl` . For example, `KUBECTL_PLUGINS_GLOBAL_FLAG_NAMESPACE` , `KUBECTL_PLUGINS_GLOBAL_FLAG_V` .
- `KUBECTL_PLUGINS_LOCAL_FLAG_*` : One environment variable for every local flag declared in the `plugin.yaml` descriptor. For example, `KUBECTL_PLUGINS_LOCAL_FLAG_HEAT` in the preceding `targaryen` example.

What's next

- Check the repository for some more examples of plugins.
- In case of any questions, feel free to reach out to the CLI SIG team.
- Binary plugins is still an alpha feature, so this is the time to contribute ideas and improvements to the codebase. We're also excited to hear about what you're planning to implement with plugins, so let us know!

Tutorials

This section of the Kubernetes documentation contains tutorials. A tutorial shows how to accomplish a goal that is larger than a single task. Typically a tutorial has several sections, each of which has a sequence of steps.

- Kubernetes Basics is an in-depth interactive tutorial that helps you understand the Kubernetes system and try out some basic Kubernetes features.
- Scalable Microservices with Kubernetes (Udacity)
- Introduction to Kubernetes (edX)
- Hello Minikube

Stateless Applications

- Running a Stateless Application Using a Deployment
- Example: PHP Guestbook application with Redis
- Using a Service to Access an Application in a Cluster
- Exposing an External IP Address to Access an Application in a Cluster

Stateful Applications

- StatefulSet Basics
- Running a Single-Instance Stateful Application
- Running a Replicated Stateful Application
- Example: WordPress and MySQL with Persistent Volumes
- Example: Deploying Cassandra with Stateful Sets
- Running ZooKeeper, A CP Distributed System

CI/CD Pipeline

- Set Up a CI/CD Pipeline with Kubernetes Part 1: Overview
- Set Up a CI/CD Pipeline with a Jenkins Pod in Kubernetes (Part 2)
- Run and Scale a Distributed Crossword Puzzle App with CI/CD on Kubernetes (Part 3)
- Set Up CI/CD for a Distributed Crossword Puzzle App on Kubernetes (Part 4)

Connecting Applications

- Connecting a Front End to a Back End Using a Service

Services

- Using Source IP

What’s next

If you would like to write a tutorial, see Using Page Templates for information about the tutorial page type and the tutorial template.

Overview

asics

ıalkthrough of the basics of the Kubernetes cluster orchestration system.
ɔme background information on major Kubernetes features and
ın interactive online tutorial. These interactive tutorials let you manage a
ntainerized applications (/docs/concepts/overview/what-is-
ners) for yourself.

ɔrials, you can learn to:

ːed application on a cluster

ıt

rized application with a new software version

rized application

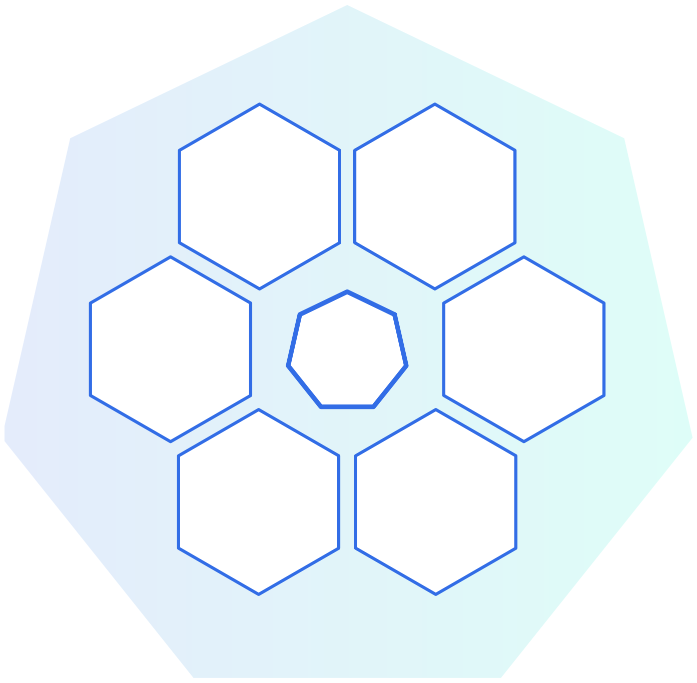
da to run a virtual terminal in your web browser that runs Minikube, a
ment of Kubernetes that can run anywhere. There's no need to install any
ything; each interactive tutorial runs directly out of your web browser

ɔernetes do for you?

es, users expect applications to be available 24/7, and developers expect
ɔf those applications several times a day. Containerization helps package
goals, enabling applications to be released and updated in an easy and
me. Kubernetes helps you make sure those containerized applications run

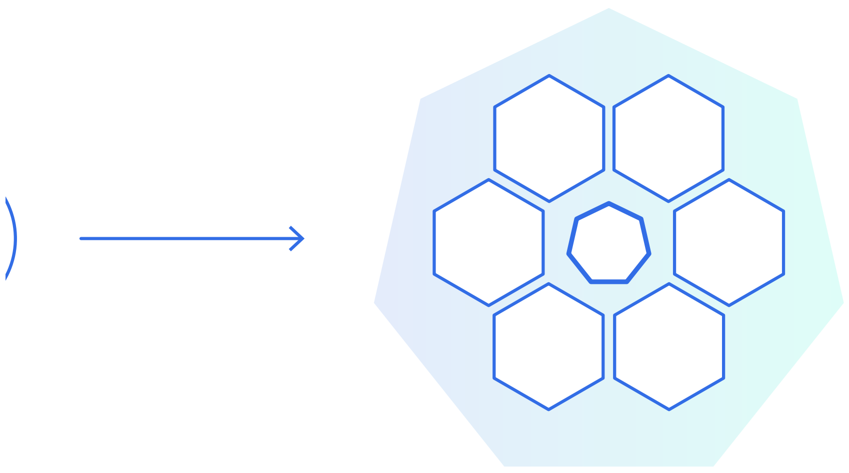
nt, and helps them find the resources and tools they need to work.
epts/overview/what-is-kubernetes/) is a production-ready, open source
Google's accumulated experience in container orchestration, combined
from the community.

asics Modules

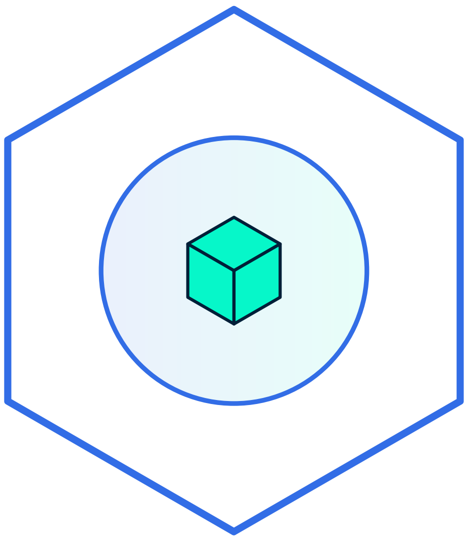


-basics/cluster-intro/)

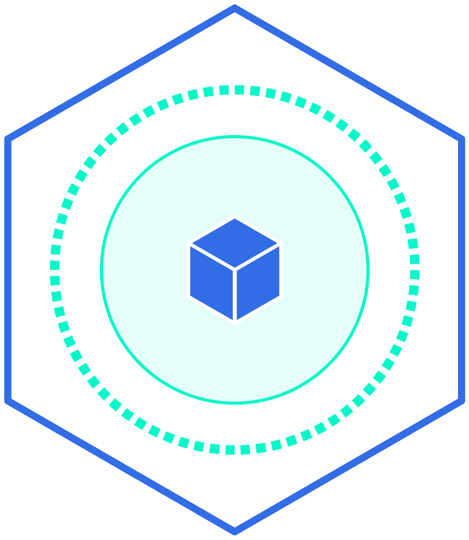
cluster



s-basics/deploy-intro/)

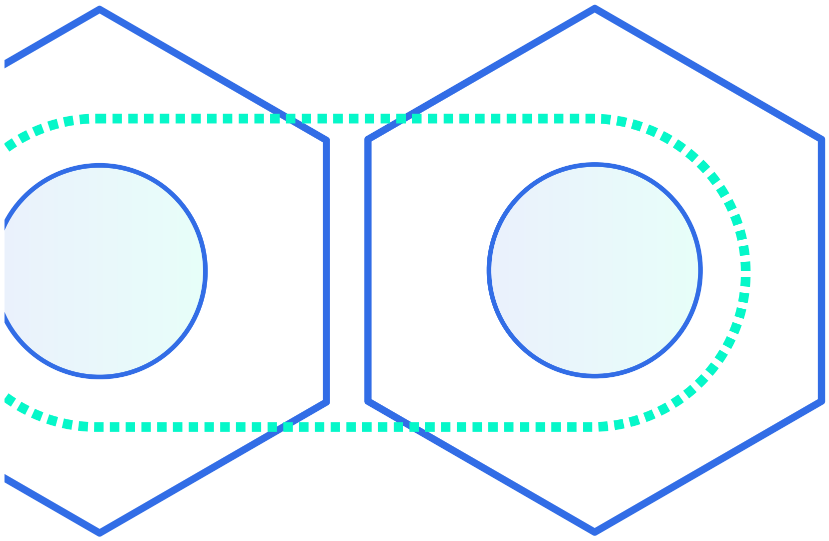


s-basics/explore-intro/)

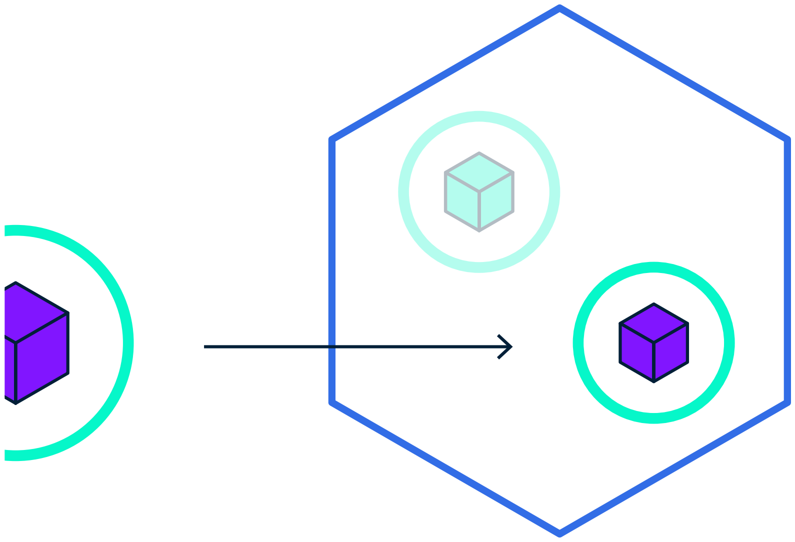


s-basics/expose-intro/)

icly



s-basics/scale-intro/)



s-basics/update-intro/)

(/docs/tutorials/kubernetes-basics/cluster-intro/)

Using Minikube to Create a Cluster

ıetes cluster is.

ɛ is.

:luster using an online terminal.

:ters

; a highly available cluster of computers that are connected to work as ctions in Kubernetes allow you to deploy containerized applications to a ṃ specifically to individual machines. To make use of this new model of ; need to be packaged in a way that decouples them from individual ɔntainerized. Containerized applications are more flexible and available models, where applications were installed directly onto specific deeply integrated into the host. **Kubernetes automates the distribution :ation containers across a cluster in a more efficient way.** Kubernetes is github.com/kubernetes/kubernetes) platform and is production-ready.

ısists of two types of resources:

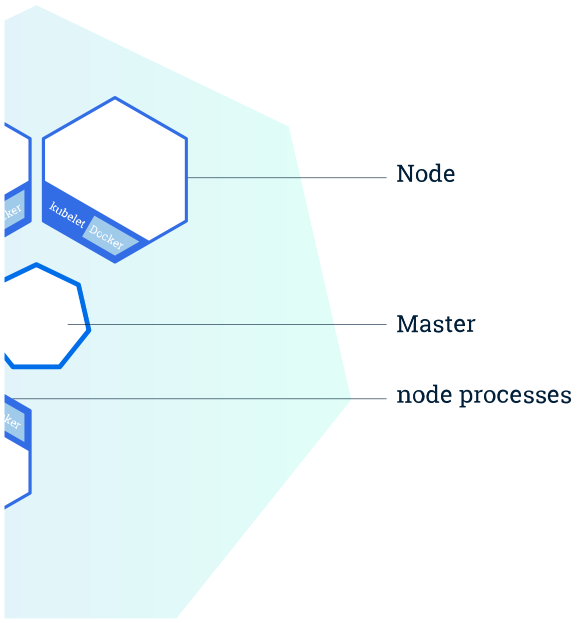
ates the cluster

ɛrs that run applications

ɛr

ction-grade, open-source platform that orchestrates the placement
ution of application containers within and across computer clusters.

am



ıetes cluster

ble for managing the cluster. The master coordinates all activities in your
ng applications, maintaining applications' desired state, scaling
out new updates.

sical computer that serves as a worker machine in a Kubernetes
ı Kubelet, which is an agent for managing the node and communicating
ster. The node should also have tools for handling container operations,
www.docker.com/) or rkt (https://coreos.com/rkt/). A Kubernetes cluster
traffic should have a minimum of three nodes.

:luster and the nodes are used to host the running applications.

ations on Kubernetes, you tell the master to start the application
schedules the containers to run on the cluster's nodes. **The nodes
naster using the Kubernetes API**, which the master exposes. End users
etes API directly to interact with the cluster.

ı be deployed on either physical or virtual machines. To get started with
t, you can use Minikube (https://github.com/kubernetes/minikube).
: Kubernetes implementation that creates a VM on your local machine
ıster containing only one node. Minikube is available for Linux, macOS,
The Minikube CLI provides basic bootstrapping operations for working
ng start, stop, status, and delete. For this tutorial, however, you'll use a
with Minikube pre-installed.

t Kubernetes is, let's go to the online tutorial and start our first cluster!

torial › (/docs/tutorials/kubernetes-basics/cluster-interactive/)

Interactive Tutorial - Creating a Cluster

2 › (/docs/tutorials/kubernetes-basics/deploy-intro/)

Using kubectl to Create a Deployment

tion Deployments.

on Kubernetes with kubectl.

loyments

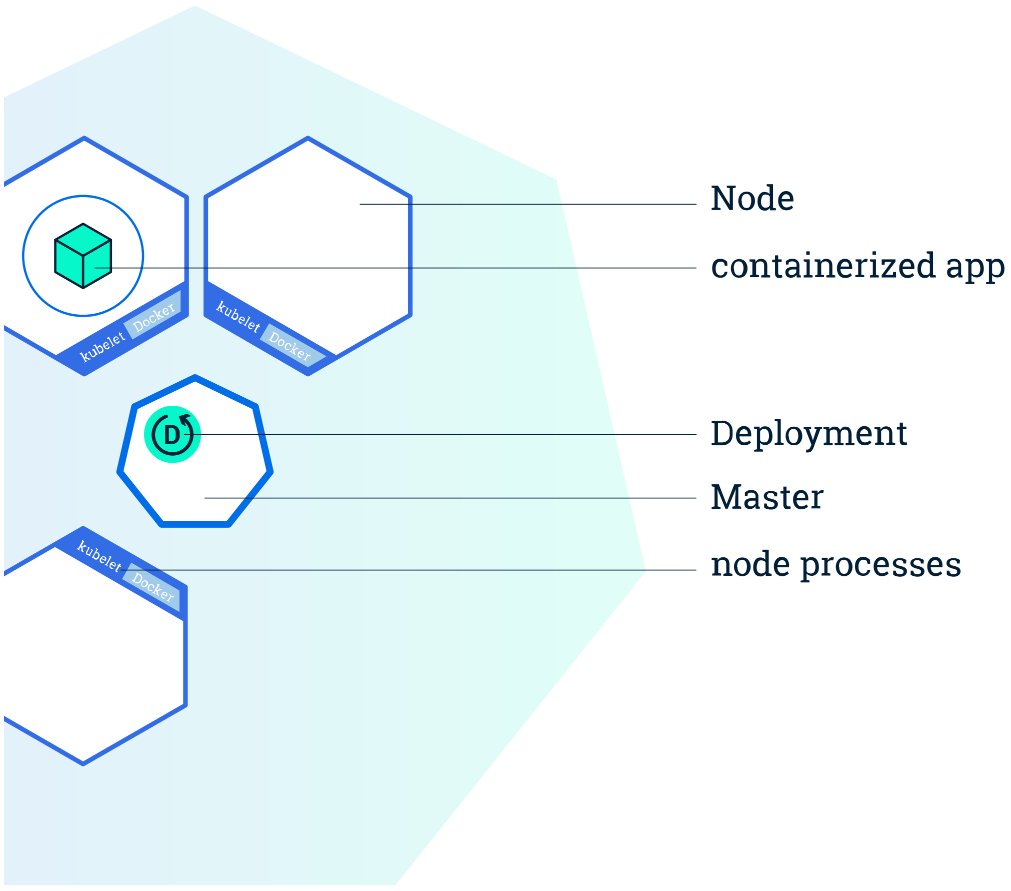
Kubernetes cluster, you can deploy your containerized applications on
eate a Kubernetes **Deployment** configuration. The Deployment instructs
e and update instances of your application. Once you've created a
etes master schedules mentioned application instances onto individual

tances are created, a Kubernetes Deployment Controller continuously
s. If the Node hosting an instance goes down or is deleted, the
places it. **This provides a self-healing mechanism to address machine**

orld, installation scripts would often be used to start applications, but they
om machine failure. By both creating your application instances and
ross Nodes, Kubernetes Deployments provide a fundamentally different
management.

ɔnsible for creating and updating instances of your application

ɹr first app on Kubernetes



Kuberneters Cluster

age a Deployment by using the Kubernetes command line interface, e Kubernetes API to interact with the cluster. In this module, you'll learn ctl commands needed to create Deployments that run your applications

›yment, you'll need to specify the container image for your application and
ıat you want to run. You can change that information later by updating
es 5 (/docs/tutorials/kubernetes-basics/scale-intro/) and 6
tes-basics/update-intro/) of the bootcamp discuss how you can scale
nents.

›e packaged into one of the supported container formats in order to be
es

, we'll use a Node.js (https://nodejs.org) application packaged in a Docker
›de and the Dockerfile are available in the GitHub repository
ernetes/kubernetes-bootcamp) for the Kubernetes Bootcamp.

t Deployments are, let's go to the online tutorial and deploy our first app!

torial › (/docs/tutorials/kubernetes-basics/deploy-interactive/)

Interactive Tutorial - Deploying an App

3 › (/docs/tutorials/kubernetes-basics/explore-intro/)

Viewing Pods and Nodes

etes Pods.

etes Nodes.

ved applications.

ods

loyment in Module 2 (/docs/tutorials/kubernetes-basics/deploy-intro/), **d** to host your application instance. A Pod is a Kubernetes abstraction of one or more application containers (such as Docker or rkt), and some se containers. Those resources include:

Volumes

ique cluster IP address

ow to run each container, such as the container image version or specific

ation-specific "logical host" and can contain different application atively tightly coupled. For example, a Pod might include both the e.js app as well as a different container that feeds the data to be webserver. The containers in a Pod share an IP Address and port space, d co-scheduled, and run in a shared context on the same Node.

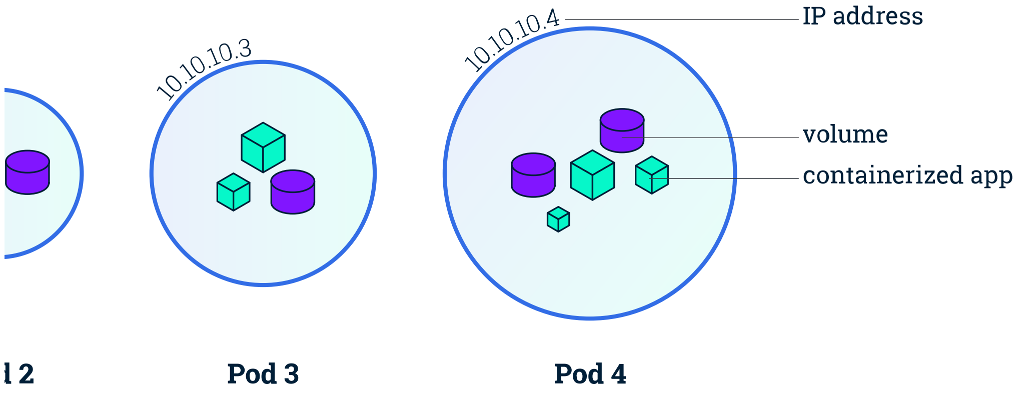
on the Kubernetes platform. When we create a Deployment on ment creates Pods with containers inside them (as opposed to creating Pod is tied to the Node where it is scheduled, and remains there until

ɔ restart policy) or deletion. In case of a Node failure, identical Pods are
able Nodes in the cluster.

ımmands

e or more application containers (such as Docker or rkt) and includes
ıes), IP address and information about how to run them.

ᴎ



Node. A Node is a worker machine in Kubernetes and may be either a
hine, depending on the cluster. Each Node is managed by the Master. A
oods, and the Kubernetes master automatically handles scheduling the
n the cluster. The Master's automatic scheduling takes into account the
ach Node.

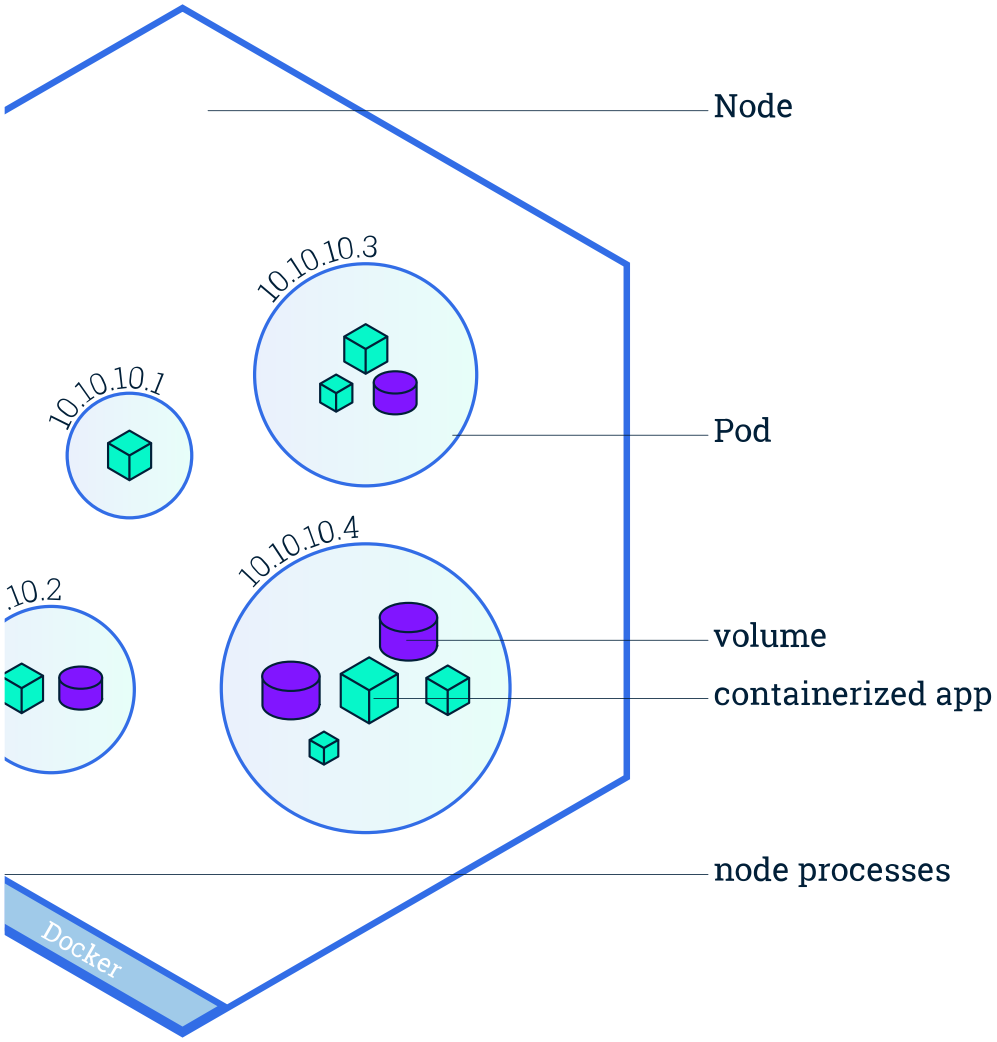
uns at least:

esponsible for communication between the Kubernetes Master and the
he Pods and the containers running on a machine.

(like Docker, rkt) responsible for pulling the container image from a
he container, and running the application.

y be scheduled together in a single Pod if they are tightly coupled and
es such as disk.

N



ing with kubectl

ials/kubernetes-basics/deploy-intro/), you used Kubectl command-line
to use it in Module 3 to get information about deployed applications and
most common operations can be done with the following kubectl

ources

how detailed information about a resource

he logs from a container in a pod

ıte a command on a container in a pod

nands to see when applications were deployed, what their current are running and what their configurations are.

about our cluster components and the command line, let's explore our

ıchine in Kubernetes and may be a VM or physical machine, depending ɐ Pods can run on one Node.

torial › (/docs/tutorials/kubernetes-basics/explore-interactive/)

Interactive Tutorial - Exploring Your App

4 › (/docs/tutorials/kubernetes-basics/expose-intro/)

Using a Service to Expose Your App

ce in Kubernetes

els and LabelSelector objects relate to a Service

on outside a Kubernetes cluster using a Service

ernetes Services

/concepts/workloads/pods/pod-overview/) are mortal. Pods in fact have
ots/workloads/pods/pod-lifecycle/). When a worker node dies, the Pods
also lost. A ReplicationController (/docs/user-guide/replication-
plicationcontroller) might then dynamically drive the cluster back to
n of new Pods to keep your application running. As another example,
essing backend with 3 replicas. Those replicas are fungible; the front-end
about backend replicas or even if a Pod is lost and recreated. That said,
s cluster has a unique IP address, even Pods on the same Node, so there
omatically reconciling changes among Pods so that your applications

is an abstraction which defines a logical set of Pods and a policy by
ervices enable a loose coupling between dependent Pods. A Service is
ferred) (/docs/concepts/configuration/overview/#general-config-tips) or
objects. The set of Pods targeted by a Service is usually determined by a
for why you might want a Service without including **`selector`** in the

a unique IP address, those IPs are not exposed outside the cluster without
your applications to receive traffic. Services can be exposed in different
`pe` in the ServiceSpec:

Exposes the Service on an internal IP in the cluster. This type makes the
ole from within the cluster.

the Service on the same port of each selected Node in the cluster using
e accessible from outside the cluster using `:`. Superset of ClusterIP.

tes an external load balancer in the current cloud (if supported) and
rnal IP to the Service. Superset of NodePort.

oses the Service using an arbitrary name (specified by **`externalName`** in
ig a CNAME record with the name. No proxy is used. This type requires
`be-dns`.

the different types of Services can be found in the Using Source IP
s/source-ip/) tutorial. Also see Connecting Applications with Services
es-networking/connect-applications-service).

ere are some use cases with Services that involve not defining **`selector`**
eated without **`selector`** will also not create the corresponding Endpoints
to manually map a Service to specific endpoints. Another possibility why
is you are strictly using **`type: ExternalName`**.

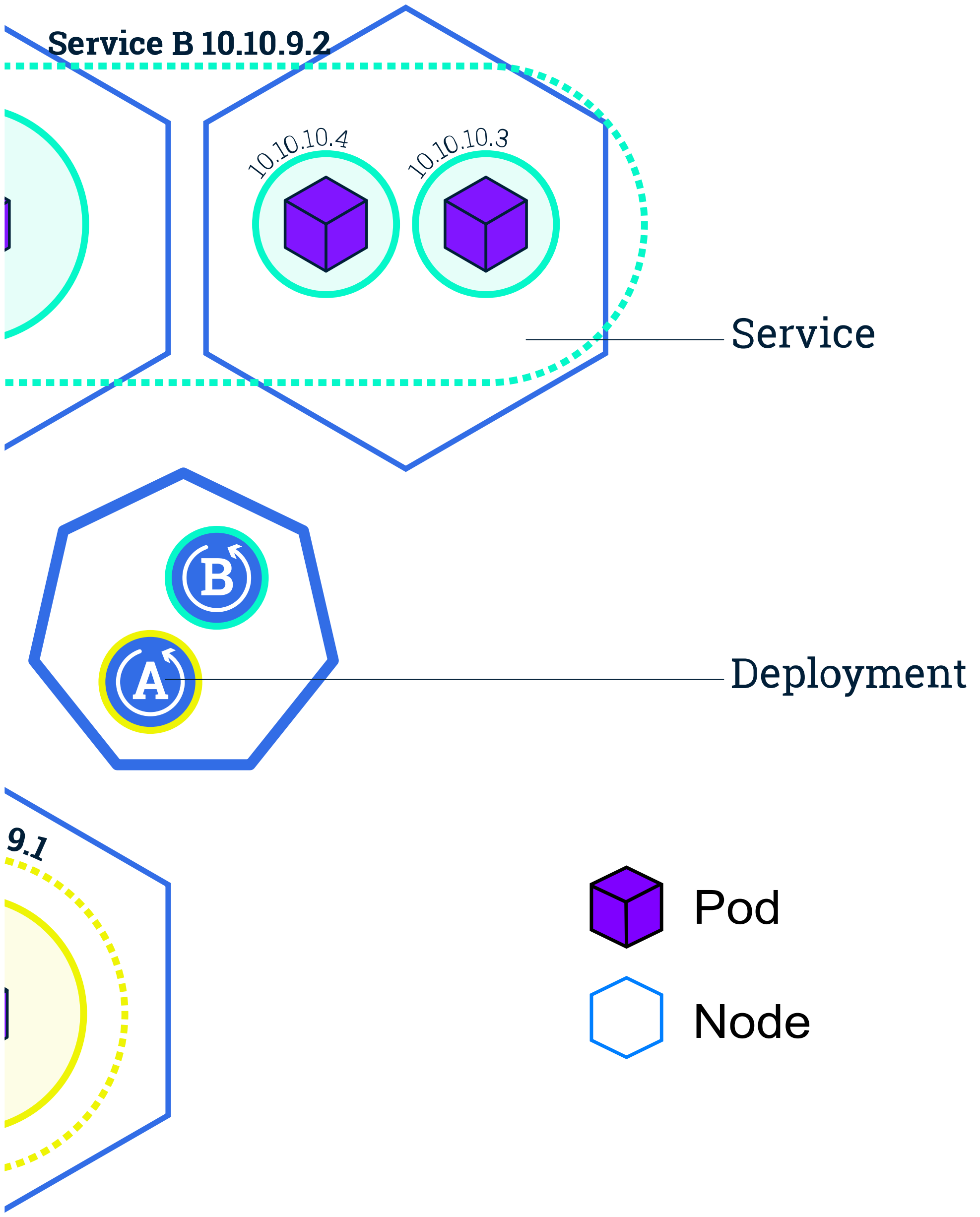
---

external traffic

affic across multiple Pods

is an abstraction layer which defines a logical set of Pods and enables
re, load balancing and service discovery for those Pods.

ɔels



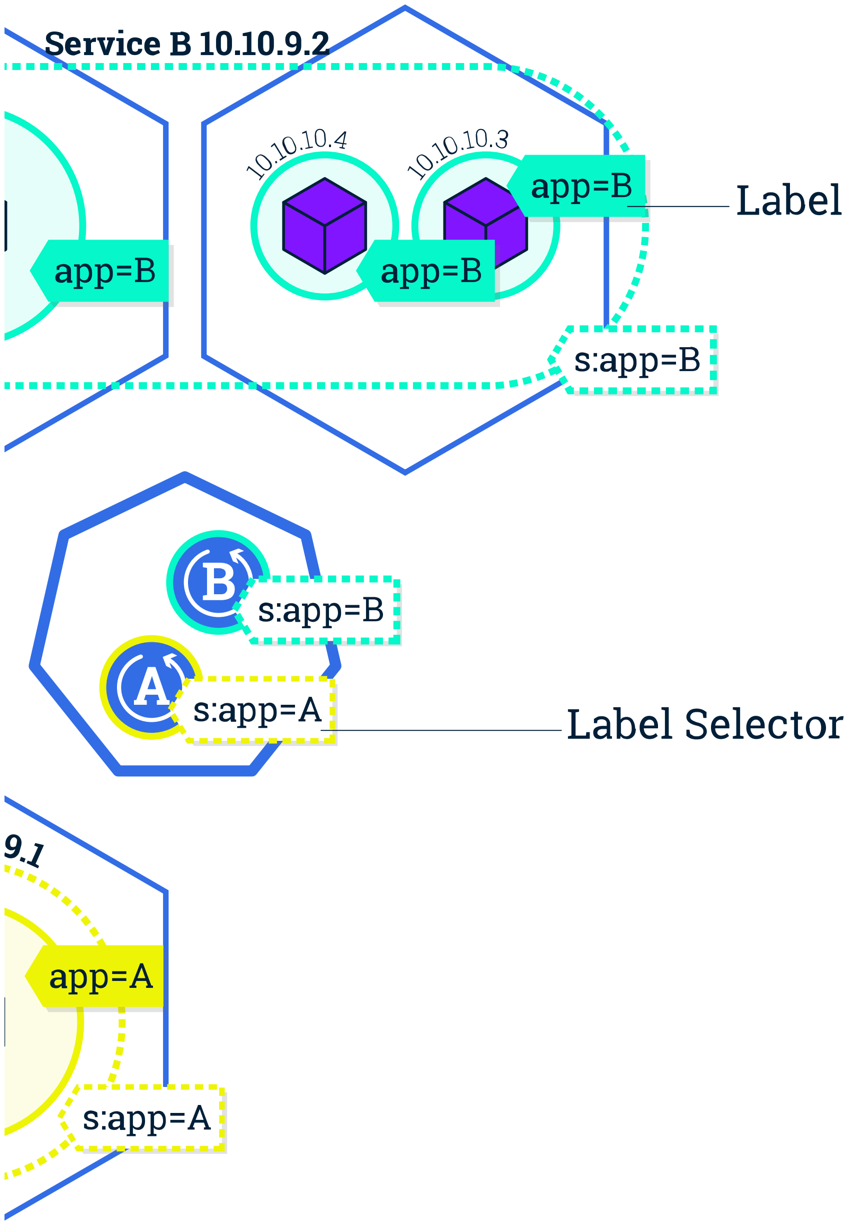
cross a set of Pods. Services are the abstraction that allow pods to die
tes without impacting your application. Discovery and routing among
s the frontend and backend components in an application) is handled by

Pods using labels and selectors (/docs/concepts/overview/working-with-
ng primitive that allows logical operation on objects in Kubernetes. Labels
hed to objects and can be used in any number of ways:

r development, test, and production

sing tags

ice at the same time you create a Deployment by using



o objects at creation time or later on. They can be modified at any time.

tion now using a Service and apply some labels.

torial › (/docs/tutorials/kubernetes-basics/expose-interactive/)

Interactive Tutorial - Exposing Your App

5 › (/docs/tutorials/kubernetes-basics/scale-intro/)

Running Multiple Instances of Your App

ubectl.

cation

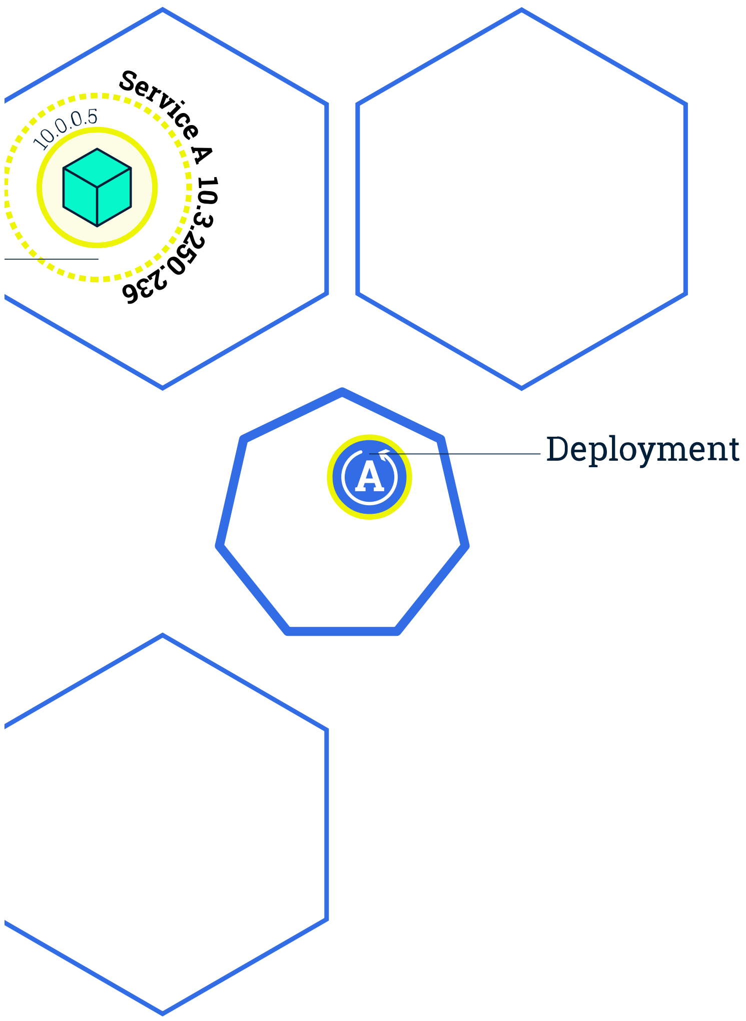
we created a Deployment
ocs/concepts/workloads/controllers/deployment/), and then exposed it
tps://kubernetes.io/docs/concepts/services-networking/service/). The
one Pod for running our application. When traffic increases, we will need
o keep up with user demand.

by changing the number of replicas in a Deployment

nent

he start a Deployment with multiple instances using the --replicas
ectl run command

iew



t will ensure new Pods are created and scheduled to Nodes with available
 will reduce the number of Pods to the new desired state. Kubernetes
ıg (http://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/) of
 the scope of this tutorial. Scaling to zero is also possible, and it will
 specified Deployment.

:es of an application will require a way to distribute the traffic to all of
ntegrated load-balancer that will distribute network traffic to all Pods of
. Services will monitor continuously the running Pods using endpoints, to
 only to available Pods.

ed by changing the number of replicas in a Deployment.

nstances of an Application running, you would be able to do Rolling
ne. We'll cover that in the next module. Now, let's go to the online terminal
ι.

torial › (/docs/tutorials/kubernetes-basics/scale-interactive/)

Interactive Tutorial - Scaling Your App

6 › (/docs/tutorials/kubernetes-basics/update-intro/)

Performing a Rolling Update

date using kubectl.

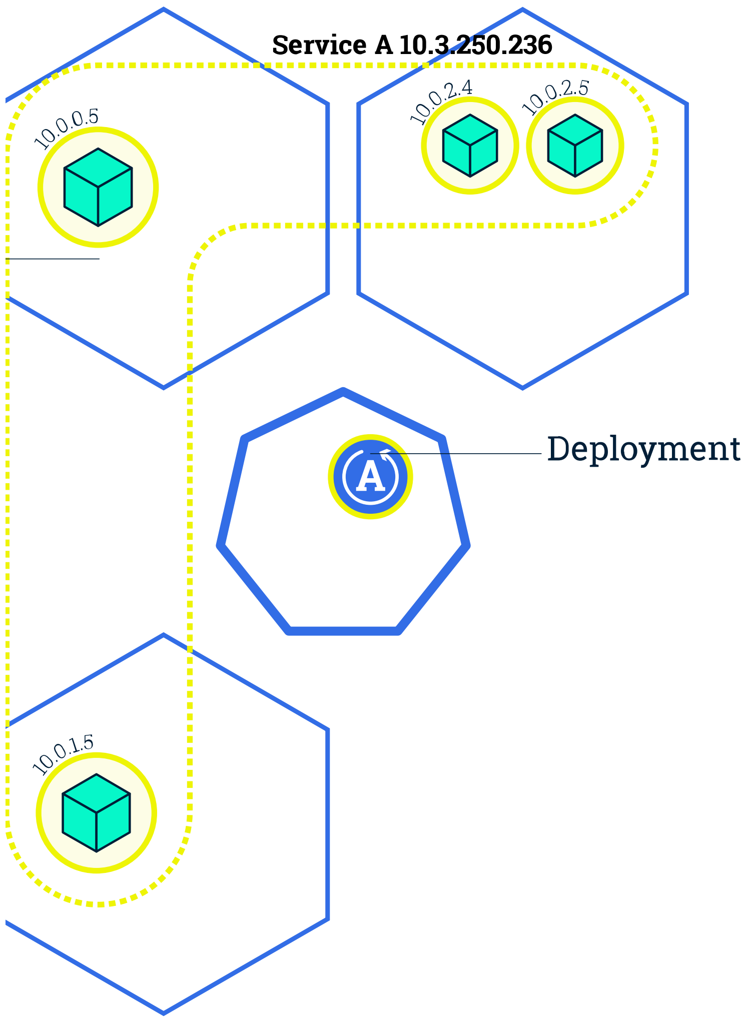
lication

s to be available all the time and developers are expected to deploy new
times a day. In Kubernetes this is done with rolling updates. **Rolling**
ents' update to take place with zero downtime by incrementally updating
ones. The new Pods will be scheduled on Nodes with available

ve scaled our application to run multiple instances. This is a requirement
without affecting application availability. By default, the maximum number
ailable during the update and the maximum number of new Pods that
oth options can be configured to either numbers or percentages (of
dates are versioned and any Deployment update can be reverted to
.

*Deployments' update to take place with zero downtime by incrementally
ces with new ones.*

es overview



aling, if a Deployment is exposed publicly, the Service will load-balance le Pods during the update. An available Pod is an instance that is the application.

e following actions:

ion from one environment to another (via container image updates) versions

ion and Continuous Delivery of applications with zero downtime

osed publicly, the Service will load-balance the traffic only to available
e.

ve tutorial, we'll update our application to a new version, and also perform

torial › (/docs/tutorials/kubernetes-basics/update-interactive/)

Interactive Tutorial - Updating Your App

Hello Minikube

The goal of this tutorial is for you to turn a simple Hello World Node.js app into an application running on Kubernetes. The tutorial shows you how to take code that you have developed on your machine, turn it into a Docker container image and then run that image on Minikube. Minikube provides a simple way of running Kubernetes on your local machine for free.

Objectives

- Run a hello world Node.js application.
- Deploy the application to Minikube.
- View application logs.
- Update the application image.

Before you begin

- For OS X, you need Homebrew to install the `xhyve` driver.
- NodeJS is required to run the sample application.
- Install Docker. On OS X, we recommend Docker for Mac.

Create a Minikube cluster

This tutorial uses Minikube to create a local cluster. This tutorial also assumes you are using Docker for Mac on OS X. If you are on a different platform like Linux, or using VirtualBox instead of Docker for Mac, the instructions to install Minikube may be slightly different. For general Minikube installation instructions, see the Minikube installation guide.

Use `curl` to download and install the latest Minikube release:

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube
  chmod +x minikube && \
  sudo mv minikube /usr/local/bin/
```

Use Homebrew to install the xhyve driver and set its permissions:

```
brew install docker-machine-driver-xhyve
sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-
sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine
```

Use Homebrew to download the `kubectl` command-line tool, which you can use to interact with Kubernetes clusters:

```
brew install kubectl
```

Determine whether you can access sites like https://cloud.google.com/container-registry/ directly without a proxy, by opening a new terminal and using

```
curl --proxy "" https://cloud.google.com/container-registry/
```

If NO proxy is required, start the Minikube cluster:

```
minikube start --vm-driver=xhyve
```

If a proxy server is required, use the following method to start Minikube cluster with proxy setting:

```
minikube start --vm-driver=xhyve --docker-env HTTP_PROXY=http://your-http-proxy-ho
```

The `--vm-driver=xhyve` flag specifies that you are using Docker for Mac. The default VM driver is VirtualBox.

Now set the Minikube context. The context is what determines which cluster `kubectl` is interacting with. You can see all your available contexts in the `~/.kube/config` file.

```
kubectl config use-context minikube
```

Verify that `kubectl` is configured to communicate with your cluster:

```
kubectl cluster-info
```

Create your Node.js application

The next step is to write the application. Save this code in a folder named `hellonode` with the filename `server.js`:

server.js

```
var http = require('http');

var handleRequest = function(request, response) {
  console.log('Received request for URL: ' + request.url);
  response.writeHead(200);
  response.end('Hello World!');
};
var www = http.createServer(handleRequest);
www.listen(8080);
```

Run your application:

```
node server.js
```

You should be able to see your "Hello World!" message at http://localhost:8080/.

Stop the running Node.js server by pressing **Ctrl-C**.

The next step is to package your application in a Docker container.

Create a Docker container image

Create a file, also in the `hellonode` folder, named `Dockerfile`. A Dockerfile describes the image that you want to build. You can build a Docker container image by extending an existing image. The image in this tutorial extends an existing Node.js image.

Dockerfile

```
FROM node:6.9.2
EXPOSE 8080
COPY server.js .
CMD node server.js
```

This recipe for the Docker image starts from the official Node.js LTS image found in the Docker registry, exposes port 8080, copies your `server.js` file to the image and starts the Node.js server.

Because this tutorial uses Minikube, instead of pushing your Docker image to a registry, you can simply build the image using the same Docker host as the Minikube VM, so that the images are automatically present. To do so, make sure you are using the Minikube Docker daemon:

```
eval $(minikube docker-env)
```

Note: Later, when you no longer wish to use the Minikube host, you can undo this change by running `eval $(minikube docker-env -u)`.

Build your Docker image, using the Minikube Docker daemon:

```
docker build -t hello-node:v1 .
```

Now the Minikube VM can run the image you built.

Create a Deployment

A Kubernetes *Pod* is a group of one or more Containers, tied together for the purposes of administration and networking. The Pod in this tutorial has only one Container. A Kubernetes *Deployment* checks on the health of your Pod and restarts the Pod's Container if it terminates. Deployments are the recommended way to manage the creation and scaling of Pods.

Use the `kubectl run` command to create a Deployment that manages a Pod. The Pod runs a Container based on your `hello-node:v1` Docker image:

```
kubectl run hello-node --image=hello-node:v1 --port=8080
```

View the Deployment:

```
kubectl get deployments
```

Output:

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node   1         1         1            1           3m
```

View the Pod:

```
kubectl get pods
```

Output:

```
NAME                         READY     STATUS    RESTARTS   AGE
hello-node-714049816-ztzrb   1/1       Running   0          6m
```

View cluster events:

```
kubectl get events
```

View the `kubectl` configuration:

```
kubectl config view
```

For more information about `kubectl` commands, see the kubectl overview.

Create a Service

By default, the Pod is only accessible by its internal IP address within the Kubernetes cluster. To make the `hello-node` Container accessible from outside the Kubernetes virtual network, you have to expose the Pod as a Kubernetes *Service*.

From your development machine, you can expose the Pod to the public internet using the `kubectl expose` command:

```
kubectl expose deployment hello-node --type=LoadBalancer
```

View the Service you just created:

```
kubectl get services
```

Output:

```
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
hello-node   10.0.0.71    <pending>     8080/TCP   6m
kubernetes   10.0.0.1     <none>        443/TCP    14d
```

The `--type=LoadBalancer` flag indicates that you want to expose your Service outside of the cluster. On cloud providers that support load balancers, an external IP address would be provisioned to access the Service. On Minikube, the `LoadBalancer` type makes the Service accessible through the `minikube service` command.

```
minikube service hello-node
```

This automatically opens up a browser window using a local IP address that serves your app and shows the “Hello World” message.

Assuming you’ve sent requests to your new web service using the browser or curl, you should now be able to see some logs:

```
kubectl logs <POD-NAME>
```

Update your app

Edit your `server.js` file to return a new message:

```
response.end('Hello World Again!');
```

Build a new version of your image:

```
docker build -t hello-node:v2 .
```

Update the image of your Deployment:

```
kubectl set image deployment/hello-node hello-node=hello-node:v2
```

Run your app again to view the new message:

```
minikube service hello-node
```

Clean up

Now you can clean up the resources you created in your cluster:

```
kubectl delete service hello-node
kubectl delete deployment hello-node
```

Optionally, stop Minikube:

```
minikube stop
```

What’s next

- Learn more about Deployment objects.
- Learn more about Deploying applications.
- Learn more about Service objects.

Configuring Redis using a ConfigMap

This page provides a real world example of how to configure Redis using a ConfigMap and builds upon the Using ConfigMap Data in Pods and Configure Containers Using a ConfigMap tasks.

Objectives

- Create a ConfigMap.
- Create a pod specification using the ConfigMap.
- Create the pod.
- Verify that the configuration was correctly applied.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- Understand Using ConfigMap Data in Pods.
- Understand Configure Containers Using a ConfigMap.

Real World Example: Configuring Redis using a ConfigMap

You can follow the steps below to configure a Redis cache using data stored in a ConfigMap.

1. Create a ConfigMap from the `docs/user-guide/configmap/redis/redis-config` file:

```
kubectl create configmap example-redis-config --from-file=docs/user-guide/conf

kubectl get configmap example-redis-config -o yaml
```

```
apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-03-30T18:14:41Z
  name: example-redis-config
  namespace: default
  resourceVersion: "24686"
  selfLink: /api/v1/namespaces/default/configmaps/example-redis-config
  uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002
```

2. Create a pod specification that uses the config data stored in the ConfigMap:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: example-redis-config
        items:
        - key: redis-config
          path: redis.conf
```

3. Create the pod:

```
kubectl create -f docs/user-guide/configmap/redis/redis-pod.yaml
```

In the example, the config volume is mounted at `/redis-master`. It uses `path` to add the `redis-config` key to a file named `redis.conf`. The file path for the redis config, therefore, is `/redis-master/redis.conf`. This is where the image will look for the config file for the redis master.

4. Use `kubectl exec` to enter the pod and run the `redis-cli` tool to verify that the configuration was correctly applied:

```
kubectl exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

What's next

- Learn more about ConfigMaps.
- See Using ConfigMap Data in Pods.

Kubernetes Object Management

The `kubectl` command-line tool supports several different ways to create and manage Kubernetes objects. This document provides an overview of the different approaches.

Management techniques

Warning: A Kubernetes object should be managed using only one technique. Mixing and matching techniques for the same object results in undefined behavior.

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1+	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate
Declarative object configuration	Directories of files	Production projects	1+	Highest

Imperative commands

When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the `kubectl` command as arguments or flags.

This is the simplest way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

Examples

Run an instance of the nginx container by creating a Deployment object:

```
kubectl run nginx --image nginx
```

Do the same thing using a different syntax:

```
kubectl create deployment nginx --image nginx
```

Trade-offs

Advantages compared to object configuration:

- Commands are simple, easy to learn and easy to remember.
- Commands require only a single step to make changes to the cluster.

Disadvantages compared to object configuration:

- Commands do not integrate with change review processes.
- Commands do not provide an audit trail associated with changes.
- Commands do not provide a source of records except for what is live.
- Commands do not provide a template for creating new objects.

Imperative object configuration

In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

See the resource reference for more details on object definitions.

Warning: The imperative `replace` command replaces the existing spec with the newly provided one, dropping all changes to the object missing from the configuration file. This approach should not be used with resource types whose specs are updated independently of the configuration file. Services of type `LoadBalancer`, for example, have their `externalIPs` field updated independently from the configuration by the cluster.

Examples

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

```
kubectl replace -f nginx.yaml
```

Trade-offs

Advantages compared to imperative commands:

- Object configuration can be stored in a source control system such as Git.
- Object configuration can integrate with processes such as reviewing changes before push and audit trails.
- Object configuration provides a template for creating new objects.

Disadvantages compared to imperative commands:

- Object configuration requires basic understanding of the object schema.

- Object configuration requires the additional step of writing a YAML file.

Advantages compared to declarative object configuration:

- Imperative object configuration behavior is simpler and easier to understand.
- As of Kubernetes version 1.5, imperative object configuration is more mature.

Disadvantages compared to declarative object configuration:

- Imperative object configuration works best on files, not directories.
- Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.

Declarative object configuration

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files. Create, update, and delete operations are automatically detected per-object by `kubectl`. This enables working on directories, where different operations might be needed for different objects.

Note: Declarative object configuration retains changes made by other writers, even if the changes are not merged back to the object configuration file. This is possible by using the `patch` API operation to write only observed differences, instead of using the `replace` API operation to replace the entire object configuration.

Examples

Process all object configuration files in the `configs` directory, and create or patch the live objects:

```
kubectl apply -f configs/
```

Recursively process directories:

```
kubectl apply -R -f configs/
```

Trade-offs

Advantages compared to imperative object configuration:

- Changes made directly to live objects are retained, even if they are not merged back into the configuration files.
- Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

- Declarative object configuration is harder to debug and understand results when they are unexpected.
- Partial updates using diffs create complex merge and patch operations.

What’s next

- Managing Kubernetes Objects Using Imperative Commands
- Managing Kubernetes Objects Using Object Configuration (Imperative)
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes Object Schema Reference

Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run`: Create a new Deployment object to run Containers in one or more Pods.
- `expose`: Create a new Service object to load balance traffic across Pods.
- `autoscale`: Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale` : Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate` : Add or remove an annotation from an object.
- `label` : Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set` : Set an aspect of an object.

Note: In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit` : Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch` : Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in API Conventions.

How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

Note: You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named nginx:

```
kubectl delete deployment/nginx
```

How to view an object

There are several commands for printing information about an object:

- `get` : Prints basic information about matching objects. Use `get -h` to see a list of options.
- `describe` : Prints aggregated detailed information about matching objects.
- `logs` : Prints the stdout and stderr for a container running in a Pod.

Using `set` commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip <myservicename> -o yaml --dry-run | kubectl set s
```

1. The `kubectl create service -o yaml --dry-run` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc -o yaml --dry-run > /tmp/srv.yaml
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

What’s next

- Managing Kubernetes Objects Using Object Configuration (Imperative)
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes Object Schema Reference

Imperative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by using the `kubectl` command-line tool along with an object configuration file written in YAML or JSON. This document explains how to define and manage objects using configuration files.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

How to create objects

You can use `kubectl create -f` to create an object from a configuration file. Refer to the kubernetes object schema reference for details.

- `kubectl create -f <filename|url>`

How to update objects

Warning: Updating objects with the `replace` command drops all parts of the spec not specified in the configuration file. This should not be used with objects whose specs are partially managed by the cluster, such as Services of type `LoadBalancer`, where the `externalIPs` field is managed independently from the configuration file. Independently managed fields must be copied to the configuration file to prevent `replace` from dropping them.

You can use `kubectl replace -f` to update a live object according to a configuration file.

- `kubectl replace -f <filename|url>`

How to delete objects

You can use `kubectl delete -f` to delete an object that is described in a configuration file.

- `kubectl delete -f <filename|url>`

How to view an object

You can use `kubectl get -f` to view information about an object that is described in a configuration file.

- `kubectl get -f <filename|url> -o yaml`

The `-o yaml` flag specifies that the full object configuration is printed. Use `kubectl get -h` to see a list of options.

Limitations

The `create`, `replace`, and `delete` commands work well when each object's configuration is fully defined and recorded in its configuration file. However when a live object is updated, and the updates are not merged into its configuration file, the updates will be lost the next time a `replace` is executed. This can happen if a controller, such as a HorizontalPodAutoscaler, makes updates directly to a live object. Here's an example:

1. You create an object from a configuration file.
2. Another source updates the object by changing some field.
3. You replace the object from the configuration file. Changes made by the other source in step 2 are lost.

If you need to support multiple writers to the same object, you can use `kubectl apply` to manage the object.

Creating and editing an object from a URL without saving the configuration

Suppose you have the URL of an object configuration file. You can use `kubectl create --edit` to make changes to the configuration before the object is created. This is particularly useful for tutorials and tasks that point to a configuration file that could be modified by the reader.

```
kubectl create -f <url> --edit
```

Migrating from imperative commands to imperative object configuration

Migrating from imperative commands to imperative object configuration involves several manual steps.

1. Export the live object to a local object configuration file:

```
kubectl get <kind>/<name> -o yaml --export > <kind>_<name>.yaml
```

2. Manually remove the status field from the object configuration file.

3. For subsequent object management, use `replace` exclusively.

```
kubectl replace -f <kind>_<name>.yaml
```

Defining controller selectors and PodTemplate labels

Warning: Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example label:

```
selector:
  matchLabels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes Object Schema Reference

Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl apply` to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files.

Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See Kubernetes Object Management for a discussion of the advantages and disadvantage of each kind of object management.

Before you begin

Declarative object configuration requires a firm understanding of the Kubernetes object definitions and configuration. Read and complete the following documents if you have not already:

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files

Following are definitions for terms used in this document:

- *object configuration file / configuration file*: A file that defines the configuration for a Kubernetes object. This topic shows how to pass configuration files to `kubectl apply`. Configuration files are typically stored in source control, such as Git.
- *live object configuration / live configuration*: The live configuration values of an object, as observed by the Kubernetes cluster. These are kept in the Kubernetes cluster storage, typically etcd.
- *declarative configuration writer / declarative writer*: A person or software component that makes updates to a live object. The live writers referred to in this topic make changes to object

configuration files and run `kubectl apply` to write the changes.

How to create objects

Use `kubectl apply` to create all objects, except those that already exist, defined by configuration files in a specified directory:

```
kubectl apply -f <directory>/
```

This sets the `kubectl.kubernetes.io/last-applied-configuration: '{...}'` annotation on each object. The annotation contains the contents of the object configuration file that was used to create the object.

Note: Add the `-R` flag to recursively process directories.

Here's an example of an object configuration file:

simple_deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_de
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_depl
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"minReadySeconds":5,"template":{"metadata":{"labels":{"app":"nginx"}
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

How to update objects

You can also use `kubectl apply` to update all objects defined in a directory, even if those objects already exist. This approach accomplishes the following:

1. Sets fields that appear in the configuration file in the live configuration.
2. Clears fields removed from the configuration file in the live configuration.

```
kubectl apply -f <directory>/
```

Note: Add the `-R` flag to recursively process directories.

Here's an example configuration file:

simple_deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_de
```

Note: For purposes of illustration, the preceding command refers to a single configuration file instead of a directory.

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_depl
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"minReadySeconds":5,"template":{"metadata":{"labels":{"app":"nginx"}
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

Directly update the `replicas` field in the live configuration by using `kubectl scale`. This does not use `kubectl apply`:

```
kubectl scale deployment/nginx-deployment --replicas 2
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_depl
```

The output shows that the `replicas` field has been set to 2, and the `last-applied-configuration` annotation does not contain a `replicas` field:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"minReadySeconds":5,"template":{"metadata":{"labels":{"app":"nginx"}
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
      # ...
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.7.9` to `nginx:1.11.9`, and delete the `minReadySeconds` field:

update_deployment.yaml

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11.9 # update the image
        ports:
        - containerPort: 80
```

Apply the changes made to the configuration file:

```
kubectl apply -f https://k8s.io/docs/tutorials/object-management-kubectl/update_de
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_depl
```

The output shows the following changes to the live configuration:

- The `replicas` field retains the value of 2 set by `kubectl scale`. This is possible because it is omitted from the configuration file.
- The `image` field has been updated to `nginx:1.11.9` from `nginx:1.7.9`.

- The `last-applied-configuration` annotation has been updated with the new image.
- The `minReadySeconds` field has been cleared.
- The `last-applied-configuration` annotation no longer contains the `minReadySeconds` field.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.11.9,
    # but does not contain the updated replicas to 2
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:1.11.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.11.9 # Set by `kubectl apply`
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

Warning: Mixing `kubectl apply` with the imperative object configuration commands `create` and `replace` is not supported. This is because `create` and `replace` do not retain the `kubectl.kubernetes.io/last-applied-configuration` that `kubectl apply` uses to compute updates.

How to delete objects

There are two approaches to delete objects managed by `kubectl apply`.

Recommended: **kubectl delete -f <filename>**

Manually deleting objects using the imperative command is the recommended approach, as it is more explicit about what is being deleted, and less likely to result in the user deleting something unintentionally:

```
kubectl delete -f <filename>
```

Alternative: **kubectl apply -f <directory/> --prune -l your=label**

Only use this if you know what you are doing.

Warning: `kubectl apply --prune` is in alpha, and backwards incompatible changes might be introduced in subsequent releases.

Warning: You must be careful when using this command, so that you do not delete objects unintentionally.

As an alternative to `kubectl delete`, you can use `kubectl apply` to identify objects to be deleted after their configuration files have been removed from the directory. Apply with `--prune` queries the API server for all objects matching a set of labels, and attempts to match the returned live object configurations against the object configuration files. If an object matches the query, and it does not have a configuration file in the directory, and it does not have a `last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory/> --prune -l <labels>
```

Important: Apply with prune should only be run against the root directory containing the object configuration files. Running against sub-directories can cause objects to be unintentionally deleted if they are returned by the label selector query specified with `-l <labels>` and do not appear in the subdirectory.

How to view an object

You can use `kubectl get` with `-o yaml` to view the configuration of a live object:

```
kubectl get -f <filename|url> -o yaml
```

How apply calculates differences and merges changes

Definition: A *patch* is an update operation that is scoped to specific fields of an object instead of the entire object. This enables updating only a specific set of fields on an object without reading the object first.

When `kubectl apply` updates the live configuration for an object, it does so by sending a patch request to the API server. The patch defines updates scoped to specific fields of the live object configuration. The `kubectl apply` command calculates this patch request using the configuration file, the live configuration, and the `last-applied-configuration` annotation stored in the live configuration.

Merge patch calculation

The `kubectl apply` command writes the contents of the configuration file to the `kubectl.kubernetes.io/last-applied-configuration` annotation. This is used to identify fields that have been removed from the configuration file and need to be cleared from the live configuration. Here are the steps used to calculate which fields should be deleted or set:

1. Calculate the fields to delete. These are the fields present in `last-applied-configuration` and missing from the configuration file.
2. Calculate the fields to add or set. These are the fields present in the configuration file whose values don't match the live configuration.

Here’s an example. Suppose this is the configuration file for a Deployment object:

update_deployment.yaml

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.11.9 # update the image
        ports:
        - containerPort: 80
```

Also, suppose this is the live configuration for the same Deployment object:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"minReadySeconds":5,"template":{"metadata":{"labels":{"app":"nginx"}
      "spec":{"containers":[{"image":"nginx:1.7.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
        - containerPort: 80
      # ...
```

Here are the merge calculations that would be performed by `kubectl apply`:

1. Calculate the fields to delete by reading values from `last-applied-configuration` and comparing them to values in the configuration file. In this example, `minReadySeconds` appears in the `last-applied-configuration` annotation, but does not appear in the configuration file. **Action:** Clear `minReadySeconds` from the live configuration.
2. Calculate the fields to set by reading values from the configuration file and comparing them to values in the live configuration. In this example, the value of `image` in the configuration file does not match the value in the live configuration. **Action:** Set the value of `image` in the live configuration.

3. Set the `last-applied-configuration` annotation to match the value of the configuration file.

4. Merge the results from 1, 2, 3 into a single patch request to the API server.

Here is the live configuration that is the result of the merge:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.11.9,
    # but does not contain the updated replicas to 2
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1beta1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"
      "spec":{"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:1.11.9","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.11.9 # Set by `kubectl apply`
        # ...
        name: nginx
        ports:
        - containerPort: 80
        # ...
      # ...
    # ...
  # ...
```

How different types of fields are merged

How a particular field in a configuration file is merged with the live configuration depends on the type of the field. There are several types of fields:

- *primitive*: A field of type string, integer, or boolean. For example, `image` and `replicas` are primitive fields. **Action:** Replace.
- *map*, also called *object*: A field of type map or a complex type that contains subfields. For example, `labels`, `annotations`, `spec` and `metadata` are all maps. **Action:** Merge elements or subfields.
- *list*: A field containing a list of items that can be either primitive types or maps. For example, `containers`, `ports`, and `args` are lists. **Action:** Varies.

When `kubectl apply` updates a map or list field, it typically does not replace the entire field, but instead updates the individual subelements. For instance, when merging the `spec` on a Deployment, the entire `spec` is not replaced. Instead the subfields of `spec`, such as `replicas`, are compared and merged.

Merging changes to primitive fields

Primitive fields are replaced or cleared.

Note: '-' is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Set live to configuration file value.
Yes	No	-	Set live to local configuration.
No	-	Yes	Clear from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes to map fields

Fields that represent maps are merged by comparing each of the subfields or elements of of the map:

Note: '-' is used for "not applicable" because the value is not used.

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Compare sub fields values.
Yes	No	-	Set live to local configuration.
No	-	Yes	Delete from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes for fields of type list

Merging changes to a list uses one of three strategies:

- Replace the list.
- Merge individual elements in a list of complex elements.
- Merge a list of primitive elements.

The choice of strategy is made on a per-field basis.

Replace the list

Treat the list the same as a primitive field. Replace or delete the entire list. This preserves ordering.

Example: Use `kubectl apply` to update the `args` field of a Container in a Pod. This sets the value of `args` in the live configuration to the value in the configuration file. Any `args` elements that had previously been added to the live configuration are lost. The order of the `args` elements defined in the configuration file is retained in the live configuration.

```
# last-applied-configuration value
    args: ["a, b"]

# configuration file value
    args: ["a", "c"]

# live configuration
    args: ["a", "b", "d"]

# result after merge
    args: ["a", "c"]
```

Explanation: The merge used the configuration file value as the new list value.

Merge individual elements of a list of complex elements:

Treat the list as a map, and treat a specific field of each element as a key. Add, delete, or update individual elements. This does not preserve ordering.

This merge strategy uses a special tag on each field called a `patchMergeKey`. The `patchMergeKey` is defined for each field in the Kubernetes source code: types.go When merging a list of maps, the field specified as the `patchMergeKey` for a given element is used like a map key for that element.

Example: Use `kubectl apply` to update the `containers` field of a PodSpec. This merges the list as though it was a map where each element is keyed by `name`.

```yaml
# last-applied-configuration value
    containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a # key: nginx-helper-a; will be deleted in result
      image: helper:1.3
    - name: nginx-helper-b # key: nginx-helper-b; will be retained
      image: helper:1.3

# configuration file value
    containers:
    - name: nginx
      image: nginx:1.11
    - name: nginx-helper-b
      image: helper:1.3
    - name: nginx-helper-c # key: nginx-helper-c; will be added in result
      image: helper:1.3

# live configuration
    containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a
      image: helper:1.3
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field will be retained
    - name: nginx-helper-d # key: nginx-helper-d; will be retained
      image: helper:1.3

# result after merge
    containers:
    - name: nginx
      image: nginx:1.10
      # Element nginx-helper-a was deleted
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field was retained
    - name: nginx-helper-c # Element was added
      image: helper:1.3
    - name: nginx-helper-d # Element was ignored
      image: helper:1.3
```

Explanation:

- The container named “nginx-helper-a” was deleted because no container named “nginx-helper-a” appeared in the configuration file.

- The container named "nginx-helper-b" retained the changes to `args` in the live configuration. `kubectl apply` was able to identify that "nginx-helper-b" in the live configuration was the same "nginx-helper-b" as in the configuration file, even though their fields had different values (no `args` in the configuration file). This is because the `patchMergeKey` field value (name) was identical in both.
- The container named "nginx-helper-c" was added because no container with that name appeared in the live configuration, but one with that name appeared in the configuration file.
- The container named "nginx-helper-d" was retained because no element with that name appeared in the last-applied-configuration.

Merge a list of primitive elements

As of Kubernetes 1.5, merging lists of primitive elements is not supported.

Note: Which of the above strategies is chosen for a given field is controlled by the `patchStrategy` tag in types.go If no `patchStrategy` is specified for a field of type list, then the list is replaced.

Default field values

The API server sets certain fields to default values in the live configuration if they are not specified when the object is created.

Here's a configuration file for a Deployment. The file does not specify `strategy` or `selector` :

simple_deployment.yaml

simple_deployment.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_de
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/docs/tutorials/object-management-kubectl/simple_depl
```

The output shows that the API server set several fields to default values in the live configuration. These fields were not specified in the configuration file.

```yaml
apiVersion: apps/v1beta1
kind: Deployment
# ...
spec:
  minReadySeconds: 5
  replicas: 1 # defaulted by apiserver
  selector:
    matchLabels: # defaulted by apiserver - derived from template.metadata.labels
      app: nginx
  strategy:
    rollingUpdate: # defaulted by apiserver - derived from strategy.type
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate # defaulted apiserver
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.7.9
        imagePullPolicy: IfNotPresent # defaulted by apiserver
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP # defaulted by apiserver
        resources: {} # defaulted by apiserver
        terminationMessagePath: /dev/termination-log # defaulted by apiserver
      dnsPolicy: ClusterFirst # defaulted by apiserver
      restartPolicy: Always # defaulted by apiserver
      securityContext: {} # defaulted by apiserver
      terminationGracePeriodSeconds: 30 # defaulted by apiserver
# ...
```

Note: Some of the fields' default values have been derived from the values of other fields that were specified in the configuration file, such as the `selector` field.

In a patch request, defaulted fields are not re-defaulted unless they are explicitly cleared as part of a patch request. This can cause unexpected behavior for fields that are defaulted based on the values of other fields. When the other fields are later changed, the values defaulted from them will not be updated unless they are explicitly cleared.

For this reason, it is recommended that certain fields defaulted by the server are explicitly defined in the configuration file, even if the desired values match the server defaults. This makes it easier to recognize conflicting values that will not be re-defaulted by the server.

Example:

```
# last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

# configuration file
spec:
  strategy:
    type: Recreate # updated value
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80

# live configuration
spec:
  strategy:
    type: RollingUpdate # defaulted value
    rollingUpdate: # defaulted value derived from type
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
# result after merge - ERROR!
spec:
  strategy:
    type: Recreate # updated value: incompatible with rollingUpdate
    rollingUpdate: # defaulted value: incompatible with "type: Recreate"
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Explanation:

1. The user creates a Deployment without defining `strategy.type`.
2. The server defaults `strategy.type` to `RollingUpdate` and defaults the `strategy.rollingUpdate` values.
3. The user changes `strategy.type` to `Recreate`. The `strategy.rollingUpdate` values remain at their defaulted values, though the server expects them to be cleared. If the `strategy.rollingUpdate` values had been defined initially in the configuration file, it would have been more clear that they needed to be deleted.
4. Apply fails because `strategy.rollingUpdate` is not cleared. The `strategy.rollingupdate` field cannot be defined with a `strategy.type` of `Recreate`.

Recommendation: These fields should be explicitly defined in the object configuration file:

- Selectors and PodTemplate labels on workloads, such as Deployment, StatefulSet, Job, DaemonSet, ReplicaSet, and ReplicationController
- Deployment rollout strategy

How to clear server-defaulted fields or fields set by other writers

As of Kubernetes 1.5, fields that do not appear in the configuration file cannot be cleared by a merge operation. Here are some workarounds:

Option 1: Remove the field by directly modifying the live object.

Note: As of Kubernetes 1.5, `kubectl edit` does not work with `kubectl apply`. Using these together will cause unexpected behavior.

Option 2: Remove the field through the configuration file.

1. Add the field to the configuration file to match the live object.
2. Apply the configuration file; this updates the annotation to include the field.
3. Delete the field from the configuration file.
4. Apply the configuration file; this deletes the field from the live object and annotation.

How to change ownership of a field between the configuration file and direct imperative writers

These are the only methods you should use to change an individual object field:

- Use `kubectl apply`.
- Write directly to the live configuration without modifying the configuration file: for example, use `kubectl scale`.

Changing the owner from a direct imperative writer to a configuration file

Add the field to the configuration file. For the field, discontinue direct updates to the live configuration that do not go through `kubectl apply`.

Changing the owner from a configuration file to a direct imperative writer

As of Kubernetes 1.5, changing ownership of a field from a configuration file to an imperative writer requires manual steps:

- Remove the field from the configuration file.
- Remove the field from the `kubectl.kubernetes.io/last-applied-configuration` annotation on the live object.

Changing management methods

Kubernetes objects should be managed using only one method at a time. Switching from one method to another is possible, but is a manual process.

Exception: It is OK to use imperative deletion with declarative management.

Migrating from imperative command management to declarative object configuration

Migrating from imperative command management to declarative object configuration involves several manual steps:

1. Export the live object to a local configuration file:

   ```
   kubectl get <kind>/<name> -o yaml --export > <kind>_<name>.yaml
   ```

2. Manually remove the `status` field from the configuration file.

   Note: This step is optional, as `kubectl apply` does not update the status field even if it is present in the configuration file.

3. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

4. Change processes to use `kubectl apply` for managing the object exclusively.

Migrating from imperative object configuration to declarative object configuration

1. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

   ```
   kubectl replace --save-config -f <kind>_<name>.yaml
   ```

2. Change processes to use `kubectl apply` for managing the object exclusively.

Defining controller selectors and PodTemplate labels

Warning: Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example:

```
selector:
  matchLabels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "extensions/v1beta1/deployment/nginx"
```

Known Issues

- Prior to Kubernetes 1.6, `kubectl apply` did not support operating on objects stored in a custom resource. For these cluster versions, you should instead use imperative object configuration.

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files

- Kubectl Command Reference
- Kubernetes Object Schema Reference

Run a Stateless Application Using a Deployment

This page shows how to run an application using a Kubernetes Deployment object.

Objectives

- Create an nginx deployment.
- Use kubectl to list information about the deployment.
- Update the deployment.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Creating and exploring an nginx deployment

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.7.9 Docker image:

deployment.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in the meta data as a uniq
      # generated from the deployment name
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

1. Create a Deployment based on the YAML file:

```
kubectl apply -f https://k8s.io/docs/tasks/run-application/deployment.yaml
```

2. Display information about the Deployment:

```
kubectl describe deployment nginx-deployment

 user@computer:~/kubernetes.github.io$ kubectl describe deployment nginx-deploy
 Name:     nginx-deployment
 Namespace:    default
 CreationTimestamp:  Tue, 30 Aug 2016 18:11:37 -0700
 Labels:     app=nginx
 Annotations:    deployment.kubernetes.io/revision=1
 Selector:   app=nginx
 Replicas:   2 desired | 2 updated | 2 total | 2 available | 0 unavailable
 StrategyType:   RollingUpdate
 MinReadySeconds:  0
 RollingUpdateStrategy:  1 max unavailable, 1 max surge
 Pod Template:
   Labels:       app=nginx
   Containers:
    nginx:
     Image:              nginx:1.7.9
     Port:               80/TCP
     Environment:        <none>
     Mounts:             <none>
   Volumes:              <none>
 Conditions:
   Type          Status  Reason
   ----          ------  ------
   Available     True    MinimumReplicasAvailable
   Progressing   True    NewReplicaSetAvailable
 OldReplicaSets:   <none>
 NewReplicaSet:    nginx-deployment-1771418926 (2/2 replicas created)
 No events.
```

3. List the pods created by the deployment:

```
kubectl get pods -l app=nginx

 NAME                                READY     STATUS    RESTARTS   AGE
 nginx-deployment-1771418926-7o5ns   1/1       Running   0          16h
 nginx-deployment-1771418926-r18az   1/1       Running   0          16h
```

4. Display information about a pod:

```
kubectl describe pod <pod-name>
```

where `<pod-name>` is the name of one of your pods.

Updating the deployment

You can update the deployment by applying a new YAML file. This YAML file specifies that the deployment should be updated to use nginx 1.8.

deployment-update.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8 # Update the version of nginx from 1.7.9 to 1.8
        ports:
        - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/docs/tutorials/stateless-application/deploymen
```

2. Watch the deployment create pods with new names and delete the old pods:

```
kubectl get pods -l app=nginx
```

Scaling the application by increasing the replica count

You can increase the number of pods in your Deployment by applying a new YAML file. This YAML file sets `replicas` to 4, which specifies that the Deployment should have four pods:

deployment-scale.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # Update the replicas from 2 to 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8
        ports:
        - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/docs/tutorials/stateless-application/deploymen
```

2. Verify that the Deployment has four pods:

```
kubectl get pods -l app=nginx
```

The output is similar to this:

```
NAME                               READY     STATUS    RESTARTS   AGE
nginx-deployment-148880595-4zdqq   1/1       Running   0          25s
nginx-deployment-148880595-6zgi1   1/1       Running   0          25s
nginx-deployment-148880595-fxcez   1/1       Running   0          2m
nginx-deployment-148880595-rwovn   1/1       Running   0          2m
```

Deleting a deployment

Delete the deployment by name:

```
kubectl delete deployment nginx-deployment
```

ReplicationControllers – the Old Way

The preferred way to create a replicated application is to use a Deployment, which in turn uses a ReplicaSet. Before the Deployment and ReplicaSet were added to Kubernetes, replicated applications were configured by using a ReplicationController.

What's next

- Learn more about Deployment objects.

Example: Deploying PHP Guestbook application with Redis

This tutorial shows you how to build and deploy a simple, multi-tier web application using Kubernetes and Docker. This example consists of the following components:

Objectives

- Start up a Redis master.
- Start up Redis slaves.
- Start up the guestbook frontend.
- Expose and view the Frontend Service.
- Clean up.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Download the following configuration files:

1. redis-master-deployment.yaml
2. redis-master-service.yaml
3. redis-slave-deployment.yaml
4. redis-slave-service.yaml
5. frontend-deployment.yaml
6. frontend-service.yaml

Start up the Redis Master

The guestbook application uses Redis to store its data. It writes its data to a Redis master instance and reads data from multiple Redis slave instances.

Creating the Redis Master Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica Redis master Pod.

1. Launch a terminal window in the directory you downloaded the manifest files.
2. Apply the Redis Master Deployment from the `redis-master-deployment.yaml` file:

```
kubectl apply -f redis-master-deployment.yaml
```

guestbook/redis-master-deployment.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: redis-master
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: gcr.io/google_containers/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

1. Query the list of Pods to verify that the Redis Master Pod is running:

```
kubectl get pods
```

The response should be similar to this:

```
NAME                           READY     STATUS    RESTARTS   AGE
redis-master-1068406935-3lswp  1/1       Running   0          28s
```

2. Run the following command to view the logs from the Redis Master Pod:

```
kubectl logs -f POD-NAME
```

> **Note:** Replace POD-NAME with the name of your Pod.

Creating the Redis Master Service

The guestbook applications needs to communicate to the Redis master to write its data. You need to apply a Service to proxy the traffic to the Redis master Pod. A Service defines a policy to access the Pods.

1. Apply the Redis Master Service from the following `redis-master-service.yaml` file:

```
kubectl apply -f redis-master-service.yaml
```

guestbook/redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

> **Note:** This manifest file creates a Service named `redis-master` with a set of labels that match the labels previously defined, so the Service routes network traffic to the Redis master Pod.

1. Query the list of Services to verify that the Redis Master Service is running:

```
kubectl get service
```

The response should be similar to this:

```
NAME           CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1      <none>        443/TCP    1m
redis-master   10.0.0.151    <none>        6379/TCP   8s
```

Start up the Redis Slaves

Although the Redis master is a single pod, you can make it highly available to meet traffic demands by adding replica Redis slaves.

Creating the Redis Slave Deployment

Deployments scale based off of the configurations set in the manifest file. In this case, the Deployment object specifies two replicas.

If there are not any replicas running, this Deployment would start the two replicas on your container cluster. Conversely, if there are more than two replicas are running, it would scale down until two replicas are running.

1. Apply the Redis Slave Deployment from the `redis-slave-deployment.yaml` file:

```
kubectl apply -f redis-slave-deployment.yaml
```

guestbook/redis-slave-deployment.yaml

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: redis-slave
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v1
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
          # service launched automatically. However, if the cluster you are using
          # does not have a built-in DNS service, you can instead
          # instead access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns' line above, an
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 6379
```

1. Query the list of Pods to verify that the Redis Slave Pods are running:

```
kubectl get pods
```

The response should be similar to this:

```
NAME                             READY     STATUS              RESTARTS   AGE
redis-master-1068406935-3lswp    1/1       Running             0          1m
redis-slave-2005841000-fpvqc     0/1       ContainerCreating   0          6s
redis-slave-2005841000-phfv9     0/1       ContainerCreating   0          6s
```

Creating the Redis Slave Service

The guestbook application needs to communicate to Redis slaves to read data. To make the Redis slaves discoverable, you need to set up a Service. A Service provides transparent load balancing to a set of Pods.

1. Apply the Redis Slave Service from the following `redis-slave-service.yaml` file:

```
kubectl apply -f redis-slave-service.yaml
```

guestbook/redis-slave-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

1. Query the list of Services to verify that the Redis Slave Service is running:

```
kubectl get services
```

The response should be similar to this:

```
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1     <none>        443/TCP    2m
redis-master   10.0.0.151   <none>        6379/TCP   1m
redis-slave    10.0.0.223   <none>        6379/TCP   6s
```

Set up and Expose the Guestbook Frontend

The guestbook application has a web frontend serving the HTTP requests written in PHP. It is configured to connect to the `redis-master` Service for write requests and the `redis-slave` service for Read requests.

Creating the Guestbook Frontend Deployment

1. Apply the frontend Deployment from the following `frontend-deployment.yaml` file:

```
kubectl apply -f frontend-deployment.yaml
```

guestbook/frontend-deployment.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
          # service launched automatically. However, if the cluster you are using
          # does not have a built-in DNS service, you can instead
          # instead access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns' line above, an
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 80
```

1. Query the list of Pods to verify that the three frontend replicas are running:

```
kubectl get pods -l app=guestbook -l tier=frontend
```

The response should be similar to this:

```
NAME                       READY     STATUS    RESTARTS   AGE
frontend-3823415956-dsvc5  1/1       Running   0          54s
frontend-3823415956-k22zn  1/1       Running   0          54s
frontend-3823415956-w9gbt  1/1       Running   0          54s
```

Creating the Frontend Service

The `redis-slave` and `redis-master` Services you applied are only accessible within the container cluster because the default type for a Service is ClusterIP. `ClusterIP` provides a single IP address for the set of Pods the Service is pointing to. This IP address is accessible only within the cluster.

If you want guests to be able to access your guestbook, you must configure the frontend Service to be externally visible, so a client can request the Service from outside the container cluster. Minikube can only expose Services through `NodePort`.

> **Note:** Some cloud providers, like Google Compute Engine or Google Container Engine, support external load balancers. If your cloud provider supports load balancers and you want to use it, simply delete or comment out `type: NodePort`, and uncomment `type: LoadBalancer`.

1. Apply the frontend Service from the following `frontend-service.yaml` file:

```
kubectl apply -f frontend-service.yaml
```

guestbook/frontend-service.yaml

guestbook/frontend-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

1. Query the list of Services to verify that the frontend Service is running:

```
kubectl get services
```

The response should be similar to this:

```
NAME           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
frontend       10.0.0.112    <nodes>       80:31323/TCP   6s
kubernetes     10.0.0.1      <none>        443/TCP        4m
redis-master   10.0.0.151    <none>        6379/TCP       2m
redis-slave    10.0.0.223    <none>        6379/TCP       1m
```

Viewing the Frontend Service via **NodePort**

If you deployed this application to Minikube or a local cluster, you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

   ```
   minikube service frontend --url
   ```

   The response should be similar to this:

   ```
   http://192.168.99.100:31323
   ```

2. Copy the IP address, and load the page in your browser to view your guestbook.

Viewing the Frontend Service via **LoadBalancer**

If you deployed the `frontend-service.yaml` manifest with type: `LoadBalancer` you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

   ```
   kubectl get service frontend
   ```

   The response should be similar to this:

   ```
   NAME       CLUSTER-IP      EXTERNAL-IP        PORT(S)        AGE
   frontend   10.51.242.136   109.197.92.229     80:32372/TCP   1m
   ```

2. Copy the External IP address, and load the page in your browser to view your guestbook.

Scale the Web Frontend

Scaling up or down is easy because your servers are defined as a Service that uses a Deployment controller.

1. Run the following command to scale up the number of frontend Pods:

```
kubectl scale deployment frontend --replicas=5
```

2. Query the list of Pods to verify the number of frontend Pods running:

```
kubectl get pods
```

The response should look similar to this:

```
NAME                            READY     STATUS    RESTARTS   AGE
frontend-3823415956-70qj5       1/1       Running   0          5s
frontend-3823415956-dsvc5       1/1       Running   0          54m
frontend-3823415956-k22zn       1/1       Running   0          54m
frontend-3823415956-w9gbt       1/1       Running   0          54m
frontend-3823415956-x2pld       1/1       Running   0          5s
redis-master-1068406935-3lswp   1/1       Running   0          56m
redis-slave-2005841000-fpvqc    1/1       Running   0          55m
redis-slave-2005841000-phfv9    1/1       Running   0          55m
```

3. Run the following command to scale down the number of frontend Pods:

```
kubectl scale deployment frontend --replicas=2
```

4. Query the list of Pods to verify the number of frontend Pods running:

```
kubectl get pods
```

The response should look similar to this:

```
NAME                           READY     STATUS    RESTARTS   AGE
frontend-3823415956-k22zn      1/1       Running   0          1h
frontend-3823415956-w9gbt      1/1       Running   0          1h
redis-master-1068406935-3lswp  1/1       Running   0          1h
redis-slave-2005841000-fpvqc   1/1       Running   0          1h
redis-slave-2005841000-phfv9   1/1       Running   0          1h
```

Cleaning up

Deleting the Deployments and Services also deletes any running Pods. Use labels to delete multiple resources with one command.

1. Run the following commands to delete all Pods, Deployments, and Services.

   ```
   kubectl delete deployment -l app=redis
   kubectl delete service -l app=redis
   kubectl delete deployment -l app=guestbook
   kubectl delete service -l app=guestbook
   ```

   The responses should be:

   ```
   deployment "redis-master" deleted
   deployment "redis-slave" deleted
   service "redis-master" deleted
   service "redis-slave" deleted
   deployment "frontend" deleted
   service "frontend" deleted
   ```

2. Query the list of Pods to verify that no Pods are running:

   ```
   kubectl get pods
   ```

The response should be this:

```
No resources found.
```

What’s next

- Complete the Kubernetes Basics Interactive Tutorials
- Use Kubernetes to create a blog using Persistant Volumes for MySQL and Wordpress
- Read more about connecting applications
- Read more about Managing Resources

Use a Service to Access an Application in a Cluster

This page shows how to create a Kubernetes Service object that external clients can use to access an application running in a cluster. The Service provides load balancing for an application that has two running instances.

Objectives

- Run two instances of a Hello World application.
- Create a Service object that exposes a node port.
- Use the Service object to access the running application.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Creating a service for an application running in two pods

1. Run a Hello World application in your cluster:

```
kubectl run hello-world --replicas=2 --labels="run=load-balancer-example" --ima
```

   The preceding command creates a Deployment object and an associated ReplicaSet object. The ReplicaSet has two Pods, each of which runs the Hello World application.

2. Display information about the Deployment:

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

3. Display information about your ReplicaSet objects:

```
kubectl get replicasets
kubectl describe replicasets
```

4. Create a Service object that exposes the deployment:

```
kubectl expose deployment hello-world --type=NodePort --name=example-service
```

5. Display information about the Service:

```
kubectl describe services example-service
```

   The output is similar to this:

```
Name:                     example-service
Namespace:                default
Labels:                   run=load-balancer-example
Annotations:              <none>
Selector:                 run=load-balancer-example
Type:                     NodePort
IP:                       10.32.0.16
Port:                     <unset> 8080/TCP
Endpoints:                10.200.1.4:8080,10.200.2.5:8080
Session Affinity:         None
Events:                   <none>
```

Make a note of the NodePort value for the service. For example, in the preceding output, the NodePort value is 31496.

6. List the pods that are running the Hello World application:

```
kubectl get pods --selector="run=load-balancer-example" --output=wide
```

The output is similar to this:

```
NAME                           READY   STATUS    ...  IP           NODE
hello-world-2895499144-bsbk5   1/1     Running   ...  10.200.1.4   worker1
hello-world-2895499144-m1pwt   1/1     Running   ...  10.200.2.5   worker2
```

7. Get the public IP address of one of your nodes that is running a Hello World pod. How you get this address depends on how you set up your cluster. For example, if you are using Minikube, you can see the node address by running `kubectl cluster-info`. If you are using Google Compute Engine instances, you can use the `gcloud compute instances list` command to see the public addresses of your nodes. For more information about this command, see the GCE documentation.

8. On your chosen node, create a firewall rule that allows TCP traffic on your node port. For example, if your Service has a NodePort value of 31568, create a firewall rule that allows TCP

traffic on port 31568. Different cloud providers offer different ways of configuring firewall rules. See the GCE documentation on firewall rules, for example.

9. Use the node address and node port to access the Hello World application:

   ```
   curl http://<public-node-ip>:<node-port>
   ```

   where `<public-node-ip>` is the public IP address of your node, and `<node-port>` is the NodePort value for your service.

   The response to a successful request is a hello message:

   ```
    Hello Kubernetes!
   ```

Using a service configuration file

As an alternative to using `kubectl expose`, you can use a service configuration file to create a Service.

Cleaning up

To delete the Service, enter this command:

```
kubectl delete services example-service
```

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

```
kubectl delete deployment hello-world
```

What's next

Learn more about connecting applications with services.

Exposing an External IP Address to Access an Application in a Cluster

This page shows how to create a Kubernetes Service object that exposes an external IP address.

Objectives

- Run five instances of a Hello World application.
- Create a Service object that exposes an external IP address.
- Use the Service object to access the running application.

Before you begin

- Install kubectl.
- Use a cloud provider like Google Container Engine or Amazon Web Services to create a Kubernetes cluster. This tutorial creates an external load balancer, which requires a cloud provider.
- Configure `kubectl` to communicate with your Kubernetes API server. For instructions, see the documentation for your cloud provider.

Creating a service for an application running in five pods

1. Run a Hello World application in your cluster:

```
kubectl run hello-world --replicas=5 --labels="run=load-balancer-example" --ima
```

   The preceding command creates a Deployment object and an associated ReplicaSet object. The ReplicaSet has five Pods, each of which runs the Hello World application.

2. Display information about the Deployment:

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

3. Display information about your ReplicaSet objects:

```
kubectl get replicasets
kubectl describe replicasets
```

4. Create a Service object that exposes the deployment:

```
kubectl expose deployment hello-world --type=LoadBalancer --name=my-service
```

5. Display information about the Service:

```
kubectl get services my-service
```

   The output is similar to this:

```
 NAME         CLUSTER-IP     EXTERNAL-IP      PORT(S)    AGE
 my-service   10.3.245.137   104.198.205.71   8080/TCP   54s
```

   Note: If the external IP address is shown as <pending>, wait for a minute and enter the same command again.

6. Display detailed information about the Service:

   ```
   kubectl describe services my-service
   ```

   The output is similar to this:

   ```
    Name:           my-service
    Namespace:      default
    Labels:         run=load-balancer-example
    Annotations:    <none>
    Selector:       run=load-balancer-example
    Type:           LoadBalancer
    IP:             10.3.245.137
    LoadBalancer Ingress:   104.198.205.71
    Port:           <unset> 8080/TCP
    NodePort:       <unset> 32377/TCP
    Endpoints:      10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...
    Session Affinity:   None
    Events:         <none>
   ```

   Make a note of the external IP address exposed by your service. In this example, the external IP address is 104.198.205.71. Also note the value of Port. In this example, the port is 8080.

7. In the preceding output, you can see that the service has several endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more. These are internal addresses of the pods that are running the Hello World application. To verify these are pod addresses, enter this command:

   ```
   kubectl get pods --output=wide
   ```

   The output is similar to this:

```
NAME                         ...  IP         NODE
hello-world-2895499144-1jaz9 ...  10.0.1.6   gke-cluster-1-default-pool-e0b8d
hello-world-2895499144-2e5uh ...  10.0.1.8   gke-cluster-1-default-pool-e0b8d
hello-world-2895499144-9m4h1 ...  10.0.0.6   gke-cluster-1-default-pool-e0b8d
hello-world-2895499144-o4z13 ...  10.0.1.7   gke-cluster-1-default-pool-e0b8d
hello-world-2895499144-segjf ...  10.0.2.5   gke-cluster-1-default-pool-e0b8d
```

8. Use the external IP address to access the Hello World application:

```
curl http://<external-ip>:<port>
```

where `<external-ip>` is the external IP address of your Service, and `<port>` is the value of `Port` in your Service description.

The response to a successful request is a hello message:

```
Hello Kubernetes!
```

Cleaning up

To delete the Service, enter this command:

```
kubectl delete services my-service
```

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

```
kubectl delete deployment hello-world
```

What's next

Learn more about connecting applications with services.

StatefulSet Basics

This tutorial provides an introduction to managing applications with StatefulSets. It demonstrates how to create, delete, scale, and update the Pods of StatefulSets.

Objectives

StatefulSets are intended to be used with stateful applications and distributed systems. However, the administration of stateful applications and distributed systems on Kubernetes is a broad, complex topic. In order to demonstrate the basic features of a StatefulSet, and not to conflate the former topic with the latter, you will deploy a simple web application using a StatefulSet.

After this tutorial, you will be familiar with the following.

- How to create a StatefulSet
- How a StatefulSet manages its Pods
- How to delete a StatefulSet
- How to scale a StatefulSet
- How to update a StatefulSet's Pods

Before you begin

Before you begin this tutorial, you should familiarize yourself with the following Kubernetes concepts.

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- PersistentVolume Provisioning
- StatefulSets
- kubectl CLI

This tutorial assumes that your cluster is configured to dynamically provision PersistentVolumes. If your cluster is not configured to do so, you will have to manually provision two 1 GiB volumes prior to starting this tutorial.

Creating a StatefulSet

Begin by creating a StatefulSet using the example below. It is similar to the example presented in the StatefulSets concept. It creates a Headless Service, `nginx`, to publish the IP addresses of Pods in the StatefulSet, `web`.

`web.yaml`

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Download the example above, and save it to a file named `web.yaml`

You will need to use two terminal windows. In the first terminal, use `kubectl get` to watch the creation of the StatefulSet's Pods.

```
kubectl get pods -w -l app=nginx
```

In the second terminal, use `kubectl create` to create the Headless Service and StatefulSet defined in `web.yaml`.

```
kubectl create -f web.yaml
service "nginx" created
statefulset "web" created
```

The command above creates two Pods, each running an NGINX webserver. Get the `nginx` Service and the `web` StatefulSet to verify that they were created successfully.

```
kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     None         <none>        80/TCP    12s

kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web       2         1         20s
```

Ordered Pod Creation

For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}. Examine the output of the `kubectl get` command in the first terminal. Eventually, the output will look like the example below.

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         19s
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0         18s
```

Notice that the `web-1` Pod is not launched until the `web-0` Pod is Running and Ready.

Pods in a StatefulSet

Pods in a StatefulSet have a unique ordinal index and a stable network identity.

Examining the Pod's Ordinal Index

Get the StatefulSet's Pods.

```
kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          1m
web-1     1/1       Running   0          1m
```

As mentioned in the StatefulSets concept, the Pods in a StatefulSet have a sticky, unique identity. This identity is based on a unique ordinal index that is assigned to each Pod by the StatefulSet controller. The Pods' names take the form `<statefulset name>-<ordinal index>`. Since the `web` StatefulSet has two replicas, it creates two Pods, `web-0` and `web-1`.

Using Stable Network Identities

Each Pod has a stable hostname based on its ordinal index. Use `kubectl exec` to execute the `hostname` command in each Pod.

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1
```

Use `kubectl run` to execute a container that provides the `nslookup` command from the `dnsutils` package. Using `nslookup` on the Pods' hostnames, you can examine their in-cluster DNS addresses.

```
kubectl run -i --tty --image busybox dns-test --restart=Never --rm /bin/sh
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.6

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.6
```

The CNAME of the headless service points to SRV records (one for each Pod that is Running and Ready). The SRV records point to A record entries that contain the Pods' IP addresses.

In one terminal, watch the StatefulSet's Pods.

```
kubectl get pod -w -l app=nginx
```

In a second terminal, use `kubectl delete` to delete all the Pods in the StatefulSet.

```
kubectl delete pod -l app=nginx
pod "web-0" deleted
pod "web-1" deleted
```

Wait for the StatefulSet to restart them, and for both Pods to transition to Running and Ready.

```
kubectl get pod -w -l app=nginx
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2s
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0         34s
```

Use `kubectl exec` and `kubectl run` to view the Pods hostnames and in-cluster DNS entries.

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1

kubectl run -i --tty --image busybox dns-test --restart=Never --rm /bin/sh
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.7

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.8
```

The Pods' ordinals, hostnames, SRV records, and A record names have not changed, but the IP addresses associated with the Pods may have changed. In the cluster used for this tutorial, they have. This is why it is important not to configure other applications to connect to Pods in a StatefulSet by IP address.

If you need to find and connect to the active members of a StatefulSet, you should query the CNAME of the Headless Service (`nginx.default.svc.cluster.local`). The SRV records associated with the CNAME will contain only the Pods in the StatefulSet that are Running and Ready.

If your application already implements connection logic that tests for liveness and readiness, you can use the SRV records of the Pods (`web-0.nginx.default.svc.cluster.local` , `web-1.nginx.default.svc.cluster.local`), as they are stable, and your application will be able to discover the Pods' addresses when they transition to Running and Ready.

Writing to Stable Storage

Get the PersistentVolumeClaims for `web-0` and `web-1`.

```
kubectl get pvc -l app=nginx
NAME        STATUS    VOLUME                                     CAPACITY   ACCESS
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi        RWO
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002   1Gi        RWO
```

The StatefulSet controller created two PersistentVolumeClaims that are bound to two PersistentVolumes. As the cluster used in this tutorial is configured to dynamically provision PersistentVolumes, the PersistentVolumes were created and bound automatically.

The NGINX webservers, by default, will serve an index file at `/usr/share/nginx/html/index.html`. The `volumeMounts` field in the StatefulSets `spec` ensures that the `/usr/share/nginx/html` directory is backed by a PersistentVolume.

Write the Pods' hostnames to their `index.html` files and verify that the NGINX webservers serve the hostnames.

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'echo $(hostname) > /usr/share/nginx

for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

Note, if you instead see 403 Forbidden responses for the above curl command, you will need to fix the permissions of the directory mounted by the `volumeMounts` (due to a bug when using hostPath volumes) with:

```
for i in 0 1; do kubectl exec web-$i -- chmod 755 /usr/share/nginx/html; done
```

before retrying the curl command above.

In one terminal, watch the StatefulSet's Pods.

```
kubectl get pod -w -l app=nginx
```

In a second terminal, delete all of the StatefulSet's Pods.

```
kubectl delete pod -l app=nginx
pod "web-0" deleted
pod "web-1" deleted
```

Examine the output of the `kubectl get` command in the first terminal, and wait for all of the Pods to transition to Running and Ready.

```
kubectl get pod -w -l app=nginx
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0           2s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0          34s
```

Verify the web servers continue to serve their hostnames.

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

Even though `web-0` and `web-1` were rescheduled, they continue to serve their hostnames because the PersistentVolumes associated with their PersistentVolumeClaims are remounted to their `volumeMounts`. No matter what node `web-0` and `web-1` are scheduled on, their PersistentVolumes will be mounted to the appropriate mount points.

Scaling a StatefulSet

Scaling a StatefulSet refers to increasing or decreasing the number of replicas. This is accomplished by updating the `replicas` field. You can use either [`kubectl scale`]() or [`kubectl patch`]() to scale a StatefulSet.

Scaling Up

In one terminal window, watch the Pods in the StatefulSet.

```
kubectl get pods -w -l app=nginx
```

In another terminal window, use `kubectl scale` to scale the number of replicas to 5.

```
kubectl scale sts web --replicas=5
statefulset "web" scaled
```

Examine the output of the `kubectl get` command in the first terminal, and wait for the three additional Pods to transition to Running and Ready.

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2h
web-1     1/1       Running   0          2h
NAME      READY     STATUS    RESTARTS   AGE
web-2     0/1       Pending   0          0s
web-2     0/1       Pending   0         0s
web-2     0/1       ContainerCreating   0         0s
web-2     1/1       Running   0         19s
web-3     0/1       Pending   0         0s
web-3     0/1       Pending   0         0s
web-3     0/1       ContainerCreating   0         0s
web-3     1/1       Running   0         18s
web-4     0/1       Pending   0         0s
web-4     0/1       Pending   0         0s
web-4     0/1       ContainerCreating   0         0s
web-4     1/1       Running   0         19s
```

The StatefulSet controller scaled the number of replicas. As with [StatefulSet creation](), the StatefulSet controller created each Pod sequentially with respect to its ordinal index, and it waited for each Pod's predecessor to be Running and Ready before launching the subsequent Pod.

Scaling Down

In one terminal, watch the StatefulSet's Pods.

```
kubectl get pods -w -l app=nginx
```

In another terminal, use `kubectl patch` to scale the StatefulSet back down to three replicas.

```
kubectl patch sts web -p '{"spec":{"replicas":3}}'
statefulset "web" patched
```

Wait for `web-4` and `web-3` to transition to Terminating.

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          3h
web-1     1/1       Running             0          3h
web-2     1/1       Running             0          55s
web-3     1/1       Running             0          36s
web-4     0/1       ContainerCreating   0          18s
NAME      READY     STATUS    RESTARTS   AGE
web-4     1/1       Running   0          19s
web-4     1/1       Terminating   0         24s
web-4     1/1       Terminating   0         24s
web-3     1/1       Terminating   0         42s
web-3     1/1       Terminating   0         42s
```

Ordered Pod Termination

The controller deleted one Pod at a time, in reverse order with respect to its ordinal index, and it waited for each to be completely shutdown before deleting the next.

Get the StatefulSet's PersistentVolumeClaims.

```
kubectl get pvc -l app=nginx
NAME        STATUS    VOLUME                                     CAPACITY   ACCESS
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi        RWO
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002   1Gi        RWO
www-web-2   Bound     pvc-e1125b27-b508-11e6-932f-42010a800002   1Gi        RWO
www-web-3   Bound     pvc-e1176df6-b508-11e6-932f-42010a800002   1Gi        RWO
www-web-4   Bound     pvc-e11bb5f8-b508-11e6-932f-42010a800002   1Gi        RWO
```

There are still five PersistentVolumeClaims and five PersistentVolumes. When exploring a Pod's stable storage, we saw that the PersistentVolumes mounted to the Pods of a StatefulSet are not deleted whenthe StatefulSet's Pods are deleted. This is still true when Pod deletion is caused by scaling the StatefulSet down.

Updating StatefulSets

In Kubernetes 1.7 and later, the StatefulSet controller supports automated updates. The strategy used is determined by the `spec.updateStrategy` field of the StatefulSet API Object. This feature can be used to upgrade the container images, resource requests and/or limits, labels, and annotations of the Pods in a StatefulSet. There are two valid update strategies, `RollingUpdate` and `OnDelete`.

Rolling Update

The `RollingUpdate` update strategy will update all Pods in a StatefulSet, in reverse ordinal order, while respecting the StatefulSet guarantees.

Patch the `web` StatefulSet to apply the `RollingUpdate` update strategy.

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate
statefulset "web" patched
```

In one terminal window, patch the `web` StatefulSet to change the container image again.

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/
statefulset "web" patched
```

In another terminal, watch the Pods in the StatefulSet.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          7m
web-1     1/1       Running   0          7m
web-2     1/1       Running   0          8m
web-2     1/1       Terminating   0         8m
web-2     1/1       Terminating   0         8m
web-2     0/1       Terminating   0         8m
web-2     0/1       Terminating   0         8m
web-2     0/1       Terminating   0         8m
web-2     0/1       Terminating   0         8m
web-2     0/1       Pending   0         0s
web-2     0/1       Pending   0         0s
web-2     0/1       ContainerCreating   0         0s
web-2     1/1       Running   0         19s
web-1     1/1       Terminating   0         8m
web-1     0/1       Terminating   0         8m
web-1     0/1       Terminating   0         8m
web-1     0/1       Terminating   0         8m
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0         6s
web-0     1/1       Terminating   0         7m
web-0     1/1       Terminating   0         7m
web-0     0/1       Terminating   0         7m
web-0     0/1       Terminating   0         7m
web-0     0/1       Terminating   0         7m
web-0     0/1       Terminating   0         7m
web-0     0/1       Pending   0         0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         10s
```

The Pods in the StatefulSet are updated in reverse ordinal order. The StatefulSet controller terminates each Pod, and waits for it to transition to Running and Ready prior to updating the next Pod. Note that, even though the StatefulSet controller will not proceed to update the next Pod until its ordinal successor is Running and Ready, it will restore any Pod that fails during the update to its current version. Pods that have already received the update will be restored to the updated version, and Pods that have not yet received the update will be restored to the previous version. In this way, the controller attempts to continue to keep the application healthy and the update consistent in the presence of intermittent failures.

Get the Pods to view their container images.

```
for p in 0 1 2; do kubectl get po web-$p --template '{{range $i, $c := .spec.conta
gcr.io/google_containers/nginx-slim:0.8
gcr.io/google_containers/nginx-slim:0.8
gcr.io/google_containers/nginx-slim:0.8
```

All the Pods in the StatefulSet are now running the previous container image.

Tip You can also use `kubectl rollout status sts/<name>` to view the status of a rolling update.

Staging an Update

You can stage an update to a StatefulSet by using the `partition` parameter of the `RollingUpdate` update strategy. A staged update will keep all of the Pods in the StatefulSet at the current version while allowing mutations to the StatefulSet's `.spec.template`.

Patch the `web` StatefulSet to add a partition to the `updateStrategy` field.

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate
statefulset "web" patched
```

Patch the StatefulSet again to change the container's image.

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/
statefulset "web" patched
```

Delete a Pod in the StatefulSet.

```
kubectl delete po web-2
pod "web-2" deleted
```

Wait for the Pod to be Running and Ready.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          4m
web-1     1/1       Running             0          4m
web-2     0/1       ContainerCreating   0          11s
web-2     1/1       Running   0         18s
```

Get the Pod's container.

```
kubectl get po web-2 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{
gcr.io/google_containers/nginx-slim:0.8
```

Notice that, even though the update strategy is `RollingUpdate` the StatefulSet controller restored the Pod with its original container. This is because the ordinal of the Pod is less than the `partition` specified by the `updateStrategy` .

Rolling Out a Canary

You can roll out a canary to test a modification by decrementing the `partition` you specified above.

Patch the StatefulSet to decrement the partition.

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate
statefulset "web" patched
```

Wait for `web-2` to be Running and Ready.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          4m
web-1     1/1       Running             0          4m
web-2     0/1       ContainerCreating   0          11s
web-2     1/1       Running   0         18s
```

Get the Pod's container.

```
kubectl get po web-2 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{
gcr.io/google_containers/nginx-slim:0.7
```

When you changed the `partition`, the StatefulSet controller automatically updated the `web-2` Pod because the Pod's ordinal was less than or equal to the `partition`.

Delete the `web-1` Pod.

```
kubectl delete po web-1
pod "web-1" deleted
```

Wait for the `web-1` Pod to be Running and Ready.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS        RESTARTS   AGE
web-0     1/1       Running       0          6m
web-1     0/1       Terminating   0          6m
web-2     1/1       Running       0          2m
web-1     0/1       Terminating   0         6m
web-1     0/1       Terminating   0         6m
web-1     0/1       Terminating   0         6m
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0         18s
```

Get the `web-1` Pods container.

```
kubectl get po web-1 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{
gcr.io/google_containers/nginx-slim:0.8
```

`web-1` was restored to its original configuration because the Pod's ordinal was less than the partition. When a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's `.spec.template` is updated. If a Pod that has an ordinal less than the partition is deleted or otherwise terminated, it will be restored to its original configuration.

Phased Roll Outs

You can perform a phased roll out (e.g. a linear, geometric, or exponential roll out) using a partitioned rolling update in a similar manner to how you rolled out a canary. To perform a phased roll out, set the `partition` to the ordinal at which you want the controller to pause the update.

The partition is currently set to `2`. Set the partition to `0`.

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate
statefulset "web" patched
```

Wait for all of the Pods in the StatefulSet to become Running and Ready.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          3m
web-1     0/1       ContainerCreating   0          11s
web-2     1/1       Running             0          2m
web-1     1/1       Running   0         18s
web-0     1/1       Terminating   0         3m
web-0     1/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Pending   0         0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         3s
```

Get the Pod's containers.

```
for p in 0 1 2; do kubectl get po web-$p --template '{{range $i, $c := .spec.conta
gcr.io/google_containers/nginx-slim:0.7
gcr.io/google_containers/nginx-slim:0.7
gcr.io/google_containers/nginx-slim:0.7
```

By moving the `partition` to `0`, you allowed the StatefulSet controller to continue the update process.

On Delete

The `OnDelete` update strategy implements the legacy (1.6 and prior) behavior, When you select this update strategy, the StatefulSet controller will not automatically update Pods when a modification is made to the StatefulSet's `.spec.template` field. This strategy can be selected by setting the `.spec.template.updateStrategy.type` to `OnDelete`.

Deleting StatefulSets

StatefulSet supports both Non-Cascading and Cascading deletion. In a Non-Cascading Delete, the StatefulSet's Pods are not deleted when the StatefulSet is deleted. In a Cascading Delete, both the StatefulSet and its Pods are deleted.

Non-Cascading Delete

In one terminal window, watch the Pods in the StatefulSet.

```
kubectl get pods -w -l app=nginx
```

Use `kubectl delete` to delete the StatefulSet. Make sure to supply the `--cascade=false` parameter to the command. This parameter tells Kubernetes to only delete the StatefulSet, and to not delete any of its Pods.

```
kubectl delete statefulset web --cascade=false
statefulset "web" deleted
```

Get the Pods to examine their status.

```
kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          6m
web-1     1/1       Running   0          7m
web-2     1/1       Running   0          5m
```

Even though `web` has been deleted, all of the Pods are still Running and Ready. Delete `web-0`.

```
kubectl delete pod web-0
pod "web-0" deleted
```

Get the StatefulSet's Pods.

```
kubectl get pods -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-1     1/1       Running   0          10m
web-2     1/1       Running   0          7m
```

As the `web` StatefulSet has been deleted, `web-0` has not been relaunched.

In one terminal, watch the StatefulSet's Pods.

```
kubectl get pods -w -l app=nginx
```

In a second terminal, recreate the StatefulSet. Note that, unless you deleted the `nginx` Service (which you should not have), you will see an error indicating that the Service already exists.

```
kubectl create -f web.yaml
statefulset "web" created
Error from server (AlreadyExists): error when creating "web.yaml": services "nginx
```

Ignore the error. It only indicates that an attempt was made to create the nginx Headless Service even though that Service already exists.

Examine the output of the `kubectl get` command running in the first terminal.

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-1     1/1       Running   0          16m
web-2     1/1       Running   0          2m
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         18s
web-2     1/1       Terminating   0         3m
web-2     0/1       Terminating   0         3m
web-2     0/1       Terminating   0         3m
web-2     0/1       Terminating   0         3m
```

When the `web` StatefulSet was recreated, it first relaunched `web-0`. Since `web-1` was already Running and Ready, when `web-0` transitioned to Running and Ready, it simply adopted this Pod. Since you recreated the StatefulSet with `replicas` equal to 2, once `web-0` had been recreated, and once `web-1` had been determined to already be Running and Ready, `web-2` was terminated.

Let's take another look at the contents of the `index.html` file served by the Pods' webservers.

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

Even though you deleted both the StatefulSet and the `web-0` Pod, it still serves the hostname originally entered into its `index.html` file. This is because the StatefulSet never deletes the PersistentVolumes associated with a Pod. When you recreated the StatefulSet and it relaunched `web-0`, its original PersistentVolume was remounted.

Cascading Delete

In one terminal window, watch the Pods in the StatefulSet.

```
kubectl get pods -w -l app=nginx
```

In another terminal, delete the StatefulSet again. This time, omit the `--cascade=false` parameter.

```
kubectl delete statefulset web
statefulset "web" deleted
```

Examine the output of the `kubectl get` command running in the first terminal, and wait for all of the Pods to transition to Terminating.

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          11m
web-1     1/1       Running   0          27m
NAME      READY     STATUS        RESTARTS   AGE
web-0     1/1       Terminating   0          12m
web-1     1/1       Terminating   0         29m
web-0     0/1       Terminating   0         12m
web-0     0/1       Terminating   0         12m
web-0     0/1       Terminating   0         12m
web-1     0/1       Terminating   0         29m
web-1     0/1       Terminating   0         29m
web-1     0/1       Terminating   0         29m
```

As you saw in the Scaling Down section, the Pods are terminated one at a time, with respect to the reverse order of their ordinal indices. Before terminating a Pod, the StatefulSet controller waits for the Pod's successor to be completely terminated.

Note that, while a cascading delete will delete the StatefulSet and its Pods, it will not delete the Headless Service associated with the StatefulSet. You must delete the `nginx` Service manually.

```
kubectl delete service nginx
service "nginx" deleted
```

Recreate the StatefulSet and Headless Service one more time.

```
kubectl create -f web.yaml
service "nginx" created
statefulset "web" created
```

When all of the StatefulSet's Pods transition to Running and Ready, retrieve the contents of their `index.html` files.

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

Even though you completely deleted the StatefulSet, and all of its Pods, the Pods are recreated with their PersistentVolumes mounted, and `web-0` and `web-1` will still serve their hostnames.

Finally delete the `web` StatefulSet and the `nginx` service.

```
kubectl delete service nginx
service "nginx" deleted

kubectl delete statefulset web
statefulset "web" deleted
```

Pod Management Policy

For some distributed systems, the StatefulSet ordering guarantees are unnecessary and/or undesirable. These systems require only uniqueness and identity. To address this, in Kubernetes 1.7, we introduced `.spec.podManagementPolicy` to the StatefulSet API Object.

OrderedReady Pod Management

`OrderedReady` pod management is the default for StatefulSets. It tells the StatefulSet controller to respect the ordering guarantees demonstrated above.

Parallel Pod Management

`Parallel` pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and not to wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod.

webp.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  podManagementPolicy: "Parallel"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Download the example above, and save it to a file named `webp.yaml`

This manifest is identical to the one you downloaded above except that the `.spec.podManagementPolicy` of the `web` StatefulSet is set to `Parallel`.

In one terminal, watch the Pods in the StatefulSet.

```
kubectl get po -l app=nginx -w
```

In another terminal, create the StatefulSet and Service in the manifest.

```
kubectl create -f webp.yaml
service "nginx" created
statefulset "web" created
```

Examine the output of the `kubectl get` command that you executed in the first terminal.

```
kubectl get po -l app=nginx -w
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-1     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         10s
web-1     1/1       Running   0         10s
```

The StatefulSet controller launched both `web-0` and `web-1` at the same time.

Keep the second terminal open, and, in another terminal window scale the StatefulSet.

```
kubectl scale statefulset/web --replicas=4
statefulset "web" scaled
```

Examine the output of the terminal where the `kubectl get` command is running.

```
web-3     0/1       Pending   0         0s
web-3     0/1       Pending   0         0s
web-3     0/1       Pending   0         7s
web-3     0/1       ContainerCreating   0         7s
web-2     1/1       Running   0         10s
web-3     1/1       Running   0         26s
```

The StatefulSet controller launched two new Pods, and it did not wait for the first to become Running and Ready prior to launching the second.

Keep this terminal open, and in another terminal delete the `web` StatefulSet.

```
kubectl delete sts web
```

Again, examine the output of the `kubectl get` command running in the other terminal.

```
web-3     1/1       Terminating   0         9m
web-2     1/1       Terminating   0         9m
web-3     1/1       Terminating   0         9m
web-2     1/1       Terminating   0         9m
web-1     1/1       Terminating   0         44m
web-0     1/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-3     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-1     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-2     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-1     0/1       Terminating   0         44m
web-1     0/1       Terminating   0         44m
web-1     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-3     0/1       Terminating   0         9m
web-3     0/1       Terminating   0         9m
web-3     0/1       Terminating   0         9m
```

The StatefulSet controller deletes all Pods concurrently, it does not wait for a Pod's ordinal successor to terminate prior to deleting that Pod.

Close the terminal where the `kubectl get` command is running and delete the `nginx` Service.

```
kubectl delete svc nginx
```

Cleaning up

You will need to delete the persistent storage media for the PersistentVolumes used in this tutorial. Follow the necessary steps, based on your environment, storage configuration, and provisioning method, to ensure that all storage is reclaimed.

Run a Single-Instance Stateful Application

This page shows you how to run a single-instance stateful application in Kubernetes using a PersistentVolume and a Deployment. The application is MySQL.

Objectives

- Create a PersistentVolume referencing a disk in your environment.
- Create a MySQL Deployment.
- Expose MySQL to other pods in the cluster at a known DNS name.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- You need to either have a dynamic PersistentVolume provisioner with a default StorageClass, or statically provision PersistentVolumes yourself to satisfy the PersistentVolumeClaims used here.

Deploy MySQL

You can run a stateful application by creating a Kubernetes Deployment and connecting it to an existing PersistentVolume using a PersistentVolumeClaim. For example, this YAML file describes a Deployment that runs MySQL and references the PersistentVolumeClaim. The file defines a volume mount for /var/lib/mysql, and then creates a PersistentVolumeClaim that looks for a 20G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

Note: The password is defined in the config yaml, and this is insecure. See Kubernetes Secrets for a secure solution.

mysql-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
          # Use secret in real usage
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

1. Deploy the contents of the YAML file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-deployment.ya
```

2. Display information about the Deployment:

```
kubectl describe deployment mysql

 Name:                 mysql
 Namespace:            default
 CreationTimestamp:    Tue, 01 Nov 2016 11:18:45 -0700
 Labels:               app=mysql
 Annotations:          deployment.kubernetes.io/revision=1
 Selector:             app=mysql
 Replicas:             1 desired | 1 updated | 1 total | 0 available | 1 unava
 StrategyType:         Recreate
```

```
MinReadySeconds:      0
Pod Template:
  Labels:      app=mysql
  Containers:
   mysql:
    Image:      mysql:5.6
    Port:       3306/TCP
    Environment:
      MYSQL_ROOT_PASSWORD:      password
    Mounts:
      /var/lib/mysql from mysql-persistent-storage (rw)
  Volumes:
   mysql-persistent-storage:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim
    ClaimName:  mysql-pv-claim
    ReadOnly:   false
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     False   MinimumReplicasUnavailable
  Progressing   True    ReplicaSetUpdated
OldReplicaSets:       <none>
NewReplicaSet:        mysql-63082529 (1/1 replicas created)
Events:
  FirstSeen    LastSeen    Count    From                    SubobjectPath    Type
  ---------    --------    -----    ----                    -------------    ----
  33s          33s         1        {deployment-controller }                 Norm
```

3. List the pods created by the Deployment:

```
kubectl get pods -l app=mysql

 NAME                   READY     STATUS    RESTARTS   AGE
 mysql-63082529-2z3ki   1/1       Running   0          3m
```

4. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim


 Name:         mysql-pv-claim
 Namespace:    default
 StorageClass:
 Status:       Bound
 Volume:       mysql-pv
 Labels:       <none>
 Annotations:    pv.kubernetes.io/bind-completed=yes
                 pv.kubernetes.io/bound-by-controller=yes
 Capacity:     20Gi
 Access Modes: RWO
 Events:       <none>
```

Accessing the MySQL instance

The preceding YAML file creates a service that allows other Pods in the cluster to access the database. The Service option `clusterIP: None` lets the Service DNS name resolve directly to the Pod's IP address. This is optimal when you have only one Pod behind a Service and you don't intend to increase the number of Pods.

Run a MySQL client to connect to the server:

```
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h my
```

This command creates a new Pod in the cluster running a MySQL client and connects it to the server through the Service. If it connects, you know your stateful MySQL database is up and running.

```
Waiting for pod default/mysql-client-274442439-zyp6i to be running, status is Pend
If you don't see a command prompt, try pressing enter.

mysql>
```

Updating

The image or any other part of the Deployment can be updated as usual with the `kubectl apply` command. Here are some precautions that are specific to stateful apps:

- Don't scale the app. This setup is for single-instance apps only. The underlying PersistentVolume can only be mounted to one Pod. For clustered stateful apps, see the StatefulSet documentation.
- Use `strategy:` `type: Recreate` in the Deployment configuration YAML file. This instructs Kubernetes to *not* use rolling updates. Rolling updates will not work, as you cannot have more than one Pod running at a time. The `Recreate` strategy will stop the first pod before creating a new one with the updated configuration.

Deleting a deployment

Delete the deployed objects by name:

```
kubectl delete deployment,svc mysql
kubectl delete pvc mysql-pv-claim
```

If you manually provisioned a PersistentVolume, you also need to manually delete it, as well as release the underlying resource. If you used a dynamic provisioner, it automatically deletes the PersistentVolume when it sees that you deleted the PersistentVolumeClaim. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resource upon deleting the PersistentVolume.

What's next

- Learn more about Deployment objects.
- Learn more about Deploying applications
- kubectl run documentation
- Volumes and Persistent Volumes

Run a Replicated Stateful Application

This page shows how to run a replicated stateful application using a StatefulSet controller. The example is a MySQL single-master topology with multiple slaves running asynchronous replication.

Note that **this is not a production configuration**. In particular, MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

Objectives

- Deploy a replicated MySQL topology with a StatefulSet controller.
- Send MySQL client traffic.
- Observe resistance to downtime.
- Scale the StatefulSet up and down.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:
- Katacoda
- Play with Kubernetes
- You need to either have a dynamic PersistentVolume provisioner with a default StorageClass, or statically provision PersistentVolumes yourself to satisfy the PersistentVolumeClaims used here.
- This tutorial assumes you are familiar with PersistentVolumes and StatefulSets, as well as other core concepts like Pods, Services, and ConfigMaps.
- Some familiarity with MySQL helps, but this tutorial aims to present general patterns that should be useful for other systems.

Deploy MySQL

The example MySQL deployment consists of a ConfigMap, two Services, and a StatefulSet.

ConfigMap

Create the ConfigMap from the following YAML configuration file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-configmap.yaml
```

mysql-configmap.yaml

mysql-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

This ConfigMap provides `my.cnf` overrides that let you independently control configuration on the MySQL master and slaves. In this case, you want the master to be able to serve replication logs to slaves and you want slaves to reject any writes that don't come via replication.

There's nothing special about the ConfigMap itself that causes different portions to apply to different Pods. Each Pod decides which portion to look at as it's initializing, based on information provided by the StatefulSet controller.

Services

Create the Services from the following YAML configuration file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-services.yaml
```

mysql-services.yaml

mysql-services.yaml

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the master: mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

The Headless Service provides a home for the DNS entries that the StatefulSet controller creates for each Pod that's part of the set. Because the Headless Service is named `mysql`, the Pods are accessible by resolving `<pod-name>.mysql` from within any other Pod in the same Kubernetes cluster and namespace.

The Client Service, called `mysql-read`, is a normal Service with its own cluster IP that distributes connections across all MySQL Pods that report being Ready. The set of potential endpoints includes the MySQL master and all slaves.

Note that only read queries can use the load-balanced Client Service. Because there is only one MySQL master, clients should connect directly to the MySQL master Pod (through its DNS entry within the Headless Service) to execute writes.

StatefulSet

Finally, create the StatefulSet from the following YAML configuration file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-statefulset.yaml
```

mysql-statefulset.yaml

```
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Generate mysql server-id from pod ordinal index.
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          echo [mysqld] > /mnt/conf.d/server-id.cnf
          # Add an offset to avoid reserved server-id=0 value.
          echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
          # Copy appropriate conf.d files from config-map to emptyDir.
          if [[ $ordinal -eq 0 ]]; then
            cp /mnt/config-map/master.cnf /mnt/conf.d/
          else
            cp /mnt/config-map/slave.cnf /mnt/conf.d/
          fi
        volumeMounts:
        - name: conf
```

```
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
      - name: clone-mysql
        image: gcr.io/google-samples/xtrabackup:1.0
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on master (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          [[ $ordinal -eq 0 ]] && exit 0
          # Clone data from previous peer.
          ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C /var/
          # Prepare the backup.
          xtrabackup --prepare --target-dir=/var/lib/mysql
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ALLOW_EMPTY_PASSWORD
          value: "1"
        ports:
        - name: mysql
          containerPort: 3306
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
        livenessProbe:
          exec:
            command: ["mysqladmin", "ping"]
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 5
        readinessProbe:
          exec:
            # Check we can execute queries over TCP (skip-networking is off).
            command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
          initialDelaySeconds: 5
          periodSeconds: 2
          timeoutSeconds: 1
      - name: xtrabackup
        image: gcr.io/google-samples/xtrabackup:1.0
        ports:
        - name: xtrabackup
          containerPort: 3307
        command:
        - bash
        - "-c"
        - |
          set -ex
          cd /var/lib/mysql

          # Determine binlog position of cloned data, if any.
          if [[ -f xtrabackup_slave_info ]]; then
            # XtraBackup already generated a partial "CHANGE MASTER TO" query
            # because we're cloning from an existing slave.
            mv xtrabackup_slave_info change_master_to.sql.in
            # Ignore xtrabackup_binlog_info in this case (it's useless).
            rm -f xtrabackup_binlog_info
          elif [[ -f xtrabackup_binlog_info ]]; then
            # We're cloning directly from master. Parse binlog position.
            [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$ ]] || exit
            rm xtrabackup_binlog_info
            echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
                  MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
          fi

          # Check if we need to complete a clone by starting replication.
          if [[ -f change_master_to.sql.in ]]; then
            echo "Waiting for mysqld to be ready (accepting connections)"
            until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

            echo "Initializing replication from clone position"
            # In case of container restart, attempt this at-most-once.
            mv change_master_to.sql.in change_master_to.sql.orig
            mysql -h 127.0.0.1 <<EOF
          $(<change_master_to.sql.orig),
            MASTER_HOST='mysql-0.mysql',
            MASTER_USER='root',
            MASTER_PASSWORD='',
            MASTER_CONNECT_RETRY=10;
          START SLAVE;
          EOF
          fi
          # Start a server to send backups when requested by peers.
          exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
            "xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 -
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
      volumes:
      - name: conf
        emptyDir: {}
      - name: config-map
        configMap:
          name: mysql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

You can watch the startup progress by running:

```
kubectl get pods -l app=mysql --watch
```

After a while, you should see all 3 Pods become Running:

```
NAME      READY     STATUS    RESTARTS   AGE
mysql-0   2/2       Running   0          2m
mysql-1   2/2       Running   0          1m
mysql-2   2/2       Running   0          1m
```

Press **Ctrl+C** to cancel the watch. If you don't see any progress, make sure you have a dynamic PersistentVolume provisioner enabled as mentioned in the prerequisites.

This manifest uses a variety of techniques for managing stateful Pods as part of a StatefulSet. The next section highlights some of these techniques to explain what happens as the StatefulSet creates

Pods.

Understanding stateful Pod initialization

The StatefulSet controller starts Pods one at a time, in order by their ordinal index. It waits until each Pod reports being Ready before starting the next one.

In addition, the controller assigns each Pod a unique, stable name of the form `<statefulset-name>-<ordinal-index>`. In this case, that results in Pods named `mysql-0`, `mysql-1`, and `mysql-2`.

The Pod template in the above StatefulSet manifest takes advantage of these properties to perform orderly startup of MySQL replication.

Generating configuration

Before starting any of the containers in the Pod spec, the Pod first runs any Init Containers in the order defined.

The first Init Container, named `init-mysql`, generates special MySQL config files based on the ordinal index.

The script determines its own ordinal index by extracting it from the end of the Pod name, which is returned by the `hostname` command. Then it saves the ordinal (with a numeric offset to avoid reserved values) into a file called `server-id.cnf` in the MySQL `conf.d` directory. This translates the unique, stable identity provided by the StatefulSet controller into the domain of MySQL server IDs, which require the same properties.

The script in the `init-mysql` container also applies either `master.cnf` or `slave.cnf` from the ConfigMap by copying the contents into `conf.d`. Because the example topology consists of a single MySQL master and any number of slaves, the script simply assigns ordinal `0` to be the master, and everyone else to be slaves. Combined with the StatefulSet controller's deployment order guarantee, this ensures the MySQL master is Ready before creating slaves, so they can begin replicating.

Cloning existing data

In general, when a new Pod joins the set as a slave, it must assume the MySQL master might already have data on it. It also must assume that the replication logs might not go all the way back to the beginning of time. These conservative assumptions are the key to allow a running StatefulSet to scale up and down over time, rather than being fixed at its initial size.

The second Init Container, named `clone-mysql`, performs a clone operation on a slave Pod the first time it starts up on an empty PersistentVolume. That means it copies all existing data from another running Pod, so its local state is consistent enough to begin replicating from the master.

MySQL itself does not provide a mechanism to do this, so the example uses a popular open-source tool called Percona XtraBackup. During the clone, the source MySQL server might suffer reduced performance. To minimize impact on the MySQL master, the script instructs each Pod to clone from the Pod whose ordinal index is one lower. This works because the StatefulSet controller always ensures Pod `N` is Ready before starting Pod `N+1`.

Starting replication

After the Init Containers complete successfully, the regular containers run. The MySQL Pods consist of a `mysql` container that runs the actual `mysqld` server, and an `xtrabackup` container that acts as a sidecar.

The `xtrabackup` sidecar looks at the cloned data files and determines if it's necessary to initialize MySQL replication on the slave. If so, it waits for `mysqld` to be ready and then executes the `CHANGE MASTER TO` and `START SLAVE` commands with replication parameters extracted from the XtraBackup clone files.

Once a slave begins replication, it remembers its MySQL master and reconnects automatically if the server restarts or the connection dies. Also, because slaves look for the master at its stable DNS name (`mysql-0.mysql`), they automatically find the master even if it gets a new Pod IP due to being rescheduled.

Lastly, after starting replication, the `xtrabackup` container listens for connections from other Pods requesting a data clone. This server remains up indefinitely in case the StatefulSet scales up, or in case the next Pod loses its PersistentVolumeClaim and needs to redo the clone.

Sending client traffic

You can send test queries to the MySQL master (hostname `mysql-0.mysql`) by running a temporary container with the `mysql:5.7` image and running the `mysql` client binary.

```
kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never --\
  mysql -h mysql-0.mysql <<EOF
CREATE DATABASE test;
CREATE TABLE test.messages (message VARCHAR(250));
INSERT INTO test.messages VALUES ('hello');
EOF
```

Use the hostname `mysql-read` to send test queries to any server that reports being Ready:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-read -e "SELECT * FROM test.messages"
```

You should get output like this:

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready:
+---------+
| message |
+---------+
| hello   |
+---------+
pod "mysql-client" deleted
```

To demonstrate that the `mysql-read` Service distributes connections across servers, you can run `SELECT @@server_id` in a loop:

```
kubectl run mysql-client-loop --image=mysql:5.7 -i -t --rm --restart=Never --\
  bash -ic "while sleep 1; do mysql -h mysql-read -e 'SELECT @@server_id,NOW()'; d
```

You should see the reported `@@server_id` change randomly, because a different endpoint might be selected upon each connection attempt:

```
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         100 | 2006-01-02 15:04:05 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         102 | 2006-01-02 15:04:06 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         101 | 2006-01-02 15:04:07 |
+-------------+---------------------+
```

You can press **Ctrl+C** when you want to stop the loop, but it's useful to keep it running in another window so you can see the effects of the following steps.

Simulating Pod and Node downtime

To demonstrate the increased availability of reading from the pool of slaves instead of a single server, keep the `SELECT @@server_id` loop from above running while you force a Pod out of the Ready state.

Break the Readiness Probe

The readiness probe for the `mysql` container runs the command `mysql -h 127.0.0.1 -e 'SELECT 1'` to make sure the server is up and able to execute queries.

One way to force this readiness probe to fail is to break that command:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql /usr/bin/mysql.off
```

This reaches into the actual container's filesystem for Pod `mysql-2` and renames the `mysql` command so the readiness probe can't find it. After a few seconds, the Pod should report one of its containers as not Ready, which you can check by running:

```
kubectl get pod mysql-2
```

Look for `1/2` in the `READY` column:

```
NAME      READY     STATUS    RESTARTS   AGE
mysql-2   1/2       Running   0          3m
```

At this point, you should see your `SELECT @@server_id` loop continue to run, although it never reports `102` anymore. Recall that the `init-mysql` script defined `server-id` as `100 + $ordinal`, so server ID `102` corresponds to Pod `mysql-2`.

Now repair the Pod and it should reappear in the loop output after a few seconds:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql.off /usr/bin/mysql
```

Delete Pods

The StatefulSet also recreates Pods if they're deleted, similar to what a ReplicaSet does for stateless Pods.

```
kubectl delete pod mysql-2
```

The StatefulSet controller notices that no `mysql-2` Pod exists anymore, and creates a new one with the same name and linked to the same PersistentVolumeClaim. You should see server ID `102` disappear from the loop output for a while and then return on its own.

Drain a Node

If your Kubernetes cluster has multiple Nodes, you can simulate Node downtime (such as when Nodes are upgraded) by issuing a drain.

First determine which Node one of the MySQL Pods is on:

```
kubectl get pod mysql-2 -o wide
```

The Node name should show up in the last column:

```
NAME      READY     STATUS    RESTARTS   AGE       IP            NODE
mysql-2   2/2       Running   0          15m       10.244.5.27   kubernetes-minion
```

Then drain the Node by running the following command, which cordons it so no new Pods may schedule there, and then evicts any existing Pods. Replace `<node-name>` with the name of the Node you found in the last step.

This might impact other applications on the Node, so it's best to **only do this in a test cluster**.

```
kubectl drain <node-name> --force --delete-local-data --ignore-daemonsets
```

Now you can watch as the Pod reschedules on a different Node:

```
kubectl get pod mysql-2 -o wide --watch
```

It should look something like this:

```
NAME      READY   STATUS          RESTARTS   AGE       IP            NODE
mysql-2   2/2     Terminating     0          15m       10.244.1.56   kubernetes-mi
[...]
mysql-2   0/2     Pending         0          0s        <none>        kubernetes-mi
mysql-2   0/2     Init:0/2        0          0s        <none>        kubernetes-mi
mysql-2   0/2     Init:1/2        0          20s       10.244.5.32   kubernetes-mi
mysql-2   0/2     PodInitializing 0          21s       10.244.5.32   kubernetes-mi
mysql-2   1/2     Running         0          22s       10.244.5.32   kubernetes-mi
mysql-2   2/2     Running         0          30s       10.244.5.32   kubernetes-mi
```

And again, you should see server ID `102` disappear from the `SELECT @@server_id` loop output for a while and then return.

Now uncordon the Node to return it to a normal state:

```
kubectl uncordon <node-name>
```

Scaling the number of slaves

With MySQL replication, you can scale your read query capacity by adding slaves. With StatefulSet, you can do this with a single command:

```
kubectl scale statefulset mysql  --replicas=5
```

Watch the new Pods come up by running:

```
kubectl get pods -l app=mysql --watch
```

Once they're up, you should see server IDs `103` and `104` start appearing in the `SELECT @@server_id` loop output.

You can also verify that these new servers have the data you added before they existed:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-3.mysql -e "SELECT * FROM test.messages"
```

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready:
+---------+
| message |
+---------+
| hello   |
+---------+
pod "mysql-client" deleted
```

Scaling back down is also seamless:

```
kubectl scale statefulset mysql --replicas=3
```

Note, however, that while scaling up creates new PersistentVolumeClaims automatically, scaling down does not automatically delete these PVCs. This gives you the choice to keep those initialized PVCs around to make scaling back up quicker, or to extract data before deleting them.

You can see this by running:

```
kubectl get pvc -l app=mysql
```

Which shows that all 5 PVCs still exist, despite having scaled the StatefulSet down to 3:

```
NAME           STATUS    VOLUME                                     CAPACITY   ACC
data-mysql-0   Bound     pvc-8acbf5dc-b103-11e6-93fa-42010a800002   10Gi       RWO
data-mysql-1   Bound     pvc-8ad39820-b103-11e6-93fa-42010a800002   10Gi       RWO
data-mysql-2   Bound     pvc-8ad69a6d-b103-11e6-93fa-42010a800002   10Gi       RWO
data-mysql-3   Bound     pvc-50043c45-b1c5-11e6-93fa-42010a800002   10Gi       RWO
data-mysql-4   Bound     pvc-500a9957-b1c5-11e6-93fa-42010a800002   10Gi       RWO
```

If you don't intend to reuse the extra PVCs, you can delete them:

```
kubectl delete pvc data-mysql-3
kubectl delete pvc data-mysql-4
```

Cleaning up

1. Cancel the `SELECT @@server_id` loop by pressing **Ctrl+C** in its terminal, or running the following from another terminal:

   ```
   kubectl delete pod mysql-client-loop --now
   ```

2. Delete the StatefulSet. This also begins terminating the Pods.

   ```
   kubectl delete statefulset mysql
   ```

3. Verify that the Pods disappear. They might take some time to finish terminating.

   ```
   kubectl get pods -l app=mysql
   ```

You'll know the Pods have terminated when the above returns:

```
No resources found.
```

4. Delete the ConfigMap, Services, and PersistentVolumeClaims.

   ```
   kubectl delete configmap,service,pvc -l app=mysql
   ```

5. If you manually provisioned PersistentVolumes, you also need to manually delete them, as well as release the underlying resources. If you used a dynamic provisioner, it automatically deletes the PersistentVolumes when it sees that you deleted the PersistentVolumeClaims. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resources upon deleting the PersistentVolumes.

What's next

- Look in the Helm Charts repository for other stateful application examples.

Example: Deploying WordPress and MySQL with Persistent Volumes

This tutorial shows you how to deploy a WordPress site and a MySQL database using Minikube. Both applications use PersistentVolumes and PersistentVolumeClaims to store data.

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator, and a PeristentVolumeClaim (PVC) is a set amount of storage in a PV. PersistentVolumes and PeristentVolumeClaims are independent from Pod lifecycles and preserve data through restarting, rescheduling, and even deleting Pods.

> **Warning:** This deployment is not suitable for production use cases, as it uses single instance WordPress and MySQL Pods. Consider using WordPress Helm Chart to deploy WordPress in production.

Objectives

- Create a PersistentVolume
- Create a Secret
- Deploy MySQL
- Deploy WordPress

- Clean up

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Download the following configuration files:

1. local-volumes.yaml
2. mysql-deployment.yaml
3. wordpress-deployment.yaml

Create a PersistentVolume

MySQL and Wordpress each use a PersistentVolume to store data. While Kubernetes supports many different types of PersistentVolumes, this tutorial covers hostPath.

Note: If you have a Kubernetes cluster running on Google Container Engine, please follow this guide.

Setting up a hostPath Volume

A `hostPath` mounts a file or directory from the host node's filesystem into your Pod.

Warning: Only use `hostPath` for developing and testing. With hostPath, your data lives on the node the Pod is scheduled onto and does not move between nodes. If a Pod dies and gets scheduled to another node in the cluster, the data is lost.

1. Launch a terminal window in the directory you downloaded the manifest files.
2. Create two PersistentVolumes from the `local-volumes.yaml` file:

```
kubectl create -f local-volumes.yaml
```

mysql-wordpress-persistent-volume/local-volumes.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-1
  labels:
    type: local
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /tmp/data/pv-1
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-2
  labels:
    type: local
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /tmp/data/pv-2
```

3. Run the following command to verify that two 20GiB PersistentVolumes are available:

```
kubectl get pv
```

The response should be like this:

```
NAME          CAPACITY    ACCESSMODES    RECLAIMPOLICY    STATUS       CLAIM      STO
local-pv-1    20Gi        RWO            Retain           Available
local-pv-2    20Gi        RWO            Retain           Available
```

Create a Secret for MySQL Password

A Secret is an object that stores a piece of sensitive data like a password or key. The manifest files are already configured to use a Secret, but you have to create your own Secret.

1. Create the Secret object from the following command:

```
kubectl create secret generic mysql-pass --from-literal=password=YOUR_PASSWORD
```

Note: Replace `YOUR_PASSWORD` with the password you want to apply.

2. Verify that the Secret exists by running the following command:

```
kubectl get secrets
```

The response should be like this:

```
NAME                  TYPE                                  DATA      AGE
mysql-pass                 Opaque                                1         42s
```

Note: To protect the Secret from exposure, neither `get` nor `describe` show its contents.

Deploy MySQL

The following manifest describes a single-instance MySQL Deployment. The MySQL container mounts the PersistentVolume at /var/lib/mysql. The `MYSQL_ROOT_PASSWORD` environment variable sets the database password from the Secret.

mysql-wordpress-persistent-volume/mysql-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

1. Deploy MySQL from the `mysql-deployment.yaml` file:

```
kubectl create -f mysql-deployment.yaml
```

2. Verify that the Pod is running by running the following command:

```
kubectl get pods
```

Note: It can take up to a few minutes for the Pod's Status to be `RUNNING`.

The response should be like this:

```
NAME                              READY     STATUS    RESTARTS   AGE
wordpress-mysql-1894417608-x5dzt  1/1       Running   0          40s
```

Deploy WordPress

The following manifest describes a single-instance WordPress Deployment and Service. It uses many of the same features like a PVC for persistent storage and a Secret for the password. But it also uses a different setting: `type: NodePort`. This setting exposes WordPress to traffic from outside of the cluster.

mysql-wordpress-persistent-volume/wordpress-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:4.8-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

1. Create a WordPress Service and Deployment from the `wordpress-deployment.yaml` file:

```
kubectl create -f wordpress-deployment.yaml
```

2. Verify that the Service is running by running the following command:

```
kubectl get services wordpress
```

The response should be like this:

```
NAME        CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
wordpress   10.0.0.89     <pending>      80:32406/TCP   4m
```

> **Note:** Minikube can only expose Services through `NodePort`.
>
> The `EXTERNAL-IP` is always `<pending>`.

3. Run the following command to get the IP Address for the WordPress Service:

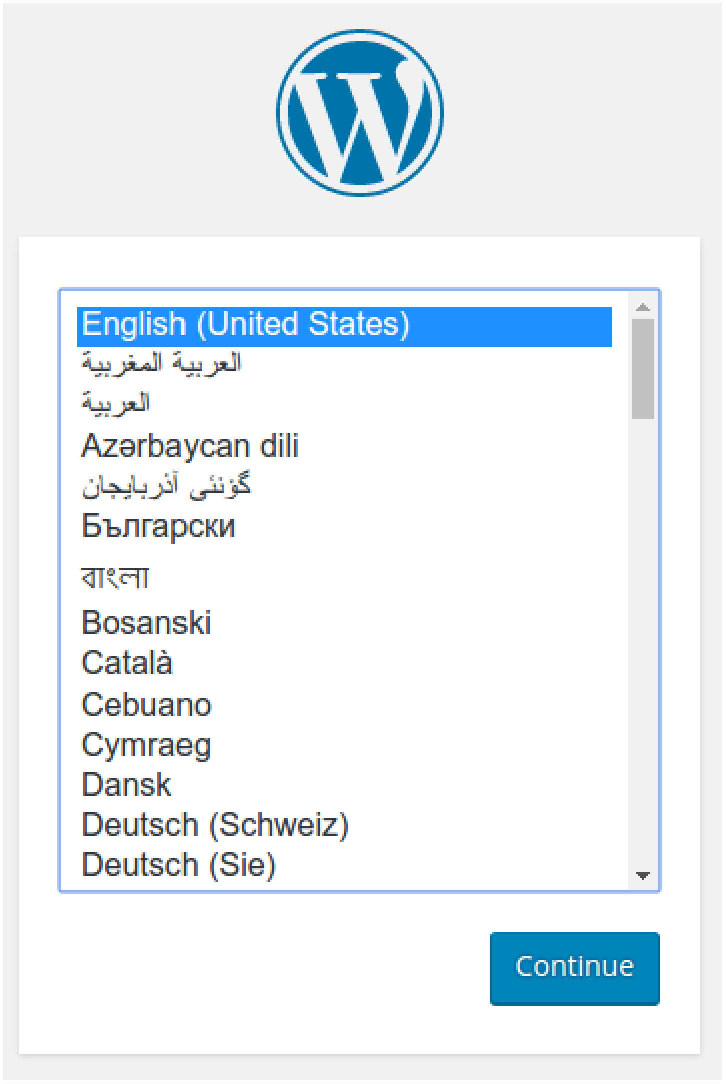
```
minikube service wordpress --url
```

The response should be like this:

```
http://1.2.3.4:32406
```

4. Copy the IP address, and load the page in your browser to view your site.

   You should see the WordPress set up page similar to the following screenshot.



Warning: Do not leave your WordPress installation on this page. If another user finds it, they can set up a website on your instance and use it to serve malicious content.

Either install WordPress by creating a username and password or delete your instance.

Cleaning up

1. Run the following command to delete your Secret:

```
kubectl delete secret mysql-pass
```

2. Run the following commands to delete all Deployments and Services:

```
kubectl delete deployment -l app=wordpress
kubectl delete service -l app=wordpress
```

3. Run the following commands to delete the PersistentVolumeClaims and the PersistentVolumes:

```
kubectl delete pvc -l app=wordpress
kubectl delete pv local-pv-1 local-pv-2
```

> **Note:** Any other Type of PersistentVolume would allow you to recreate the Deployments and Services at this point without losing data, but `hostPath` loses the data as soon as the Pod stops running.

What's next

- Learn more about Introspection and Debugging
- Learn more about Jobs
- Learn more about Port Forwarding
- Learn how to Get a Shell to a Container

Example: Deploying Cassandra with Stateful Sets

This tutorial shows you how to develop a native cloud Cassandra deployment on Kubernetes. In this instance, a custom Cassandra `SeedProvider` enables Cassandra to discover new Cassandra nodes as they join the cluster.

Deploying stateful distributed applications, like Cassandra, within a clustered environment can be challenging. StatefulSets greatly simplify this process. Please read about StatefulSets for more information about the features used in this tutorial.

Cassandra Docker

The Pods use the `gcr.io/google-samples/cassandra:v12` image from Google's container registry. The docker is based on `debian:jessie` and includes OpenJDK 8. This image includes a standard Cassandra installation from the Apache Debian repo. By using environment variables you can change values that are inserted into `cassandra.yaml`.

ENV VAR	DEFAULT VALUE
CASSANDRA_CLUSTER_NAME	'Test Cluster'
CASSANDRA_NUM_TOKENS	32
CASSANDRA_RPC_ADDRESS	0.0.0.0

Objectives

- Create and Validate a Cassandra headless Services.
- Use a StatefulSet to create a Cassandra ring.
- Validate the StatefulSet.
- Modify the StatefulSet.
- Delete the StatefulSet and its Pods.

Before you begin

To complete this tutorial, you should already have a basic familiarity with Pods, Services, and StatefulSets. In addition, you should:

- Install and Configure the `kubectl` command line
- Download cassandra-service.yaml and cassandra-statefulset.yaml
- Have a supported Kubernetes Cluster running

Note: Please read the getting started guides if you do not already have a cluster.

Additional Minikube Setup Instructions

Caution: Minikube defaults to 1024MB of memory and 1 CPU which results in an insufficient resource errors during this tutorial.

To avoid these errors, run minikube with:

```
minikube start --memory 5120 --cpus=4
```

Creating a Cassandra Headless Service

A Kubernetes Service describes a set of Pods that perform the same task.

The following `Service` is used for DNS lookups between Cassandra Pods and clients within the Kubernetes Cluster.

1. Launch a terminal window in the directory you downloaded the manifest files.
2. Create a `Service` to track all Cassandra StatefulSet Nodes from the `cassandra-service.yaml` file:

   ```
   kubectl create -f cassandra-service.yaml
   ```

cassandra/cassandra-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

Validating (optional)

Get the Cassandra `Service`.

```
kubectl get svc cassandra
```

The response should be

```
NAME        CLUSTER-IP    EXTERNAL-IP    PORT(S)     AGE
cassandra   None          <none>         9042/TCP    45s
```

If anything else returns, the service was not successfully created. Read Debug Services for common issues.

Using a StatefulSet to Create a Cassandra Ring

The StatefulSet manifest, included below, creates a Cassandra ring that consists of three Pods.

> **Note:** This example uses the default provisioner for Minikube. Please update the following StatefulSet for the cloud you are working with.

1. Update the StatefulSet if necessary.
2. Create the Cassandra StatefulSet from the `cassandra-statefulset.yaml` file:

   ```
   kubectl create -f cassandra-statefulset.yaml
   ```

cassandra/cassandra-statefulset.yaml

```
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
```

```
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v12
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
           cpu: "500m"
           memory: 1Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "PID=$(pidof java) && kill $PID && while
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: CASSANDRA_SEEDS
            value: "cassandra-0.cassandra.default.svc.cluster.local"
          - name: CASSANDRA_CLUSTER_NAME
            value: "K8Demo"
          - name: CASSANDRA_DC
            value: "DC1-K8Demo"
          - name: CASSANDRA_RACK
            value: "Rack1-K8Demo"
          - name: CASSANDRA_AUTO_BOOTSTRAP
            value: "false"
          - name: POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
        readinessProbe:
          exec:
            command:
            - /bin/bash
            - -c
            - /ready-probe.sh
          initialDelaySeconds: 15
          timeoutSeconds: 5
        # These volume mounts are persistent. They are like inline claims,
        # but not exactly because the names need to match exactly one of
        # the stateful pod volumes.
        volumeMounts:
        - name: cassandra-data
          mountPath: /cassandra_data
  # These are converted to volume claims by the controller
  # and mounted at the paths mentioned above.
  # do not use these in production until ssd GCEPersistentDisk or other ssd pd
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
      annotations:
        volume.beta.kubernetes.io/storage-class: fast
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: fast
provisioner: k8s.io/minikube-hostpath
parameters:
  type: pd-ssd
```

Validating The Cassandra StatefulSet

1. Get the Cassandra StatefulSet:

   ```
   kubectl get statefulset cassandra
   ```

   The response should be

   ```
   NAME        DESIRED   CURRENT   AGE
   cassandra   3         0         13s
   ```

The StatefulSet resource deploys Pods sequentially.

2. Get the Pods to see the ordered creation status:

```
kubectl get pods -l="app=cassandra"
```

The response should be

```
NAME          READY     STATUS              RESTARTS   AGE
cassandra-0   1/1       Running             0          1m
cassandra-1   0/1       ContainerCreating   0          8s
```

Note: It can take up to ten minutes for all three Pods to deploy.

Once all Pods are deployed, the same command returns:

```
NAME          READY     STATUS    RESTARTS   AGE
cassandra-0   1/1       Running   0          10m
cassandra-1   1/1       Running   0          9m
cassandra-2   1/1       Running   0          8m
```

3. Run the Cassandra utility nodetool to display the status of the ring.

```
kubectl exec cassandra-0 -- nodetool status
```

The response is:

```
Datacenter: DC1-K8Demo
======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID
UN  172.17.0.5  83.57 KiB  32           74.0%             e2dd09e6-d9d3-477e-9
UN  172.17.0.4  101.04 KiB  32           58.8%             f89d6835-3a42-4419-
UN  172.17.0.6  84.74 KiB  32           67.1%             a6a1e8c2-3dc5-4417-b
```

Modifying the Cassandra StatefulSet

Use `kubectl edit` to modify the size of of a Cassandra StatefulSet.

1. Run the following command:

```
kubectl edit statefulset cassandra
```

This command opens an editor in your terminal. The line you need to change is the `replicas` field.

Note: The following sample is an excerpt of the StatefulSet file.

```
yaml # Please edit the object below. Lines beginning with a '#' will be ignored, # and an empty file will abort the edit. If an error occurs while saving this file will be # reopened with the relevant failures. # apiVersion: apps/v1beta2 kind: StatefulSet metadata: creationTimestamp: 2016-08-13T18:40:58Z generation: 1 labels: app: cassandra name: cassandra namespace: default resourceVersion: "323" selfLink: /apis/apps/v1beta1/namespaces/default/statefulsets/cassandra uid: 7a219483-6185-11e6-a910-42010a8a0fc0 spec: replicas: 3
```

2. Change the number of replicas to 4, and then save the manifest.

The StatefulSet now contains 4 Pods.

3. Get the Cassandra StatefulSet to verify:

```
kubectl get statefulset cassandra
```

The response should be

```
NAME        DESIRED   CURRENT   AGE
cassandra   4         4         36m
```

Cleaning up

Deleting or scaling a StatefulSet down does not delete the volumes associated with the StatefulSet. This ensures safety first: your data is more valuable than an auto purge of all related StatefulSet resources.

Warning: Depending on the storage class and reclaim policy, deleting the Persistent Volume Claims may cause the associated volumes to also be deleted. Never assume you'll be able to access data if its volume claims are deleted.

1. Run the following commands to delete everything in a `StatefulSet`:

```
grace=$(kubectl get po cassandra-0 -o=jsonpath='{.spec.terminationGracePeriodS
  && kubectl delete statefulset -l app=cassandra \
  && echo "Sleeping $grace" \
  && sleep $grace \
  && kubectl delete pvc -l app=cassandra
```

2. Run the following command to delete the Cassandra `Service`.

```
kubectl delete service -l app=cassandra
```

What’s next

- Learn how to Scale a StatefulSet.
- Learn more about the KubernetesSeedProvider
- See more custom Seed Provider Configurations

Running ZooKeeper, A CP Distributed System

This tutorial demonstrates Apache Zookeeper on Kubernetes using StatefulSets, PodDisruptionBudgets, and PodAntiAffinity.

Objectives

After this tutorial, you will know the following.

- How to deploy a ZooKeeper ensemble using StatefulSet.
- How to consistently configure the ensemble using ConfigMaps.
- How to spread the deployment of ZooKeeper servers in the ensemble.

- How to use PodDisruptionBudgets to ensure service availability during planned maintenance.

Before you begin

Before starting this tutorial, you should be familiar with the following Kubernetes concepts.

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- PersistentVolume Provisioning
- StatefulSets
- PodDisruptionBudgets
- PodAntiAffinity
- kubectl CLI

You will require a cluster with at least four nodes, and each node will require at least 2 CPUs and 4 GiB of memory. In this tutorial you will cordon and drain the cluster's nodes. **This means that all Pods on the cluster's nodes will be terminated and evicted, and the nodes will, temporarily, become unschedulable.** You should use a dedicated cluster for this tutorial, or you should ensure that the disruption you cause will not interfere with other tenants.

This tutorial assumes that your cluster is configured to dynamically provision PersistentVolumes. If your cluster is not configured to do so, you will have to manually provision three 20 GiB volumes prior to starting this tutorial.

ZooKeeper Basics

Apache ZooKeeper is a distributed, open-source coordination service for distributed applications. ZooKeeper allows you to read, write, and observe updates to data. Data are organized in a file system like hierarchy and replicated to all ZooKeeper servers in the ensemble (a set of ZooKeeper servers).

All operations on data are atomic and sequentially consistent. ZooKeeper ensures this by using the Zab consensus protocol to replicate a state machine across all servers in the ensemble.

The ensemble uses the Zab protocol to elect a leader, and data can not be written until a leader is elected. Once a leader is elected, the ensemble uses Zab to ensure that all writes are replicated to a quorum before they are acknowledged and made visible to clients. Without respect to weighted quorums, a quorum is a majority component of the ensemble containing the current leader. For instance, if the ensemble has three servers, a component that contains the leader and one other server constitutes a quorum. If the ensemble can not achieve a quorum, data can not be written.

ZooKeeper servers keep their entire state machine in memory, but every mutation is written to a durable WAL (Write Ahead Log) on storage media. When a server crashes, it can recover its previous state by replaying the WAL. In order to prevent the WAL from growing without bound, ZooKeeper servers will periodically snapshot their in memory state to storage media. These snapshots can be loaded directly into memory, and all WAL entries that preceded the snapshot may be safely discarded.

Creating a ZooKeeper Ensemble

The manifest below contains a Headless Service, a Service, a PodDisruptionBudget, and a StatefulSet.

zookeeper.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-hs
  labels:
    app: zk
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
  clusterIP: None
  selector:
    app: zk
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-cs
  labels:
    app: zk
spec:
  ports:
  - port: 2181
    name: client
  selector:
    app: zk
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  selector:
    matchLabels:
      app: zk
  maxUnavailable: 1
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: zk
spec:
  selector:
    matchLabels:
      app: zk
  serviceName: zk-hs
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: Parallel
  template:
    metadata:
      labels:
        app: zk
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - zk
              topologyKey: "kubernetes.io/hostname"
      containers:
      - name: kubernetes-zookeeper
        imagePullPolicy: Always
        image: "gcr.io/google_containers/kubernetes-zookeeper:1.0-3.4.10"
        resources:
          requests:
            memory: "1Gi"
            cpu: "0.5"
        ports:
        - containerPort: 2181
          name: client
        - containerPort: 2888
          name: server
        - containerPort: 3888
          name: leader-election
        command:
        - sh
        - -c
        - "start-zookeeper \
          --servers=3 \
          --data_dir=/var/lib/zookeeper/data \
          --data_log_dir=/var/lib/zookeeper/data/log \
          --conf_dir=/opt/zookeeper/conf \
          --client_port=2181 \
          --election_port=3888 \
          --server_port=2888 \
          --tick_time=2000 \
          --init_limit=10 \
          --sync_limit=5 \
          --heap=512M \
          --max_client_cnxns=60 \
          --snap_retain_count=3 \
          --purge_interval=12 \
          --max_session_timeout=40000 \
          --min_session_timeout=4000 \
          --log_level=INFO"
        readinessProbe:
          exec:
            command:
            - sh
            - -c
            - "zookeeper-ready 2181"
          initialDelaySeconds: 10
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
            - sh
            - -c
            - "zookeeper-ready 2181"
          initialDelaySeconds: 10
          timeoutSeconds: 5
          timeoutSeconds: 5
        volumeMounts:
        - name: datadir
          mountPath: /var/lib/zookeeper
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
  volumeClaimTemplates:
  - metadata:
      name: datadir
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Open a command terminal, and use `kubectl apply` to create the manifest.

```
kubectl apply -f https://k8s.io/docs/tutorials/stateful-application/zookeeper.yaml
```

This creates the `zk-hs` Headless Service, the `zk-cs` Service, the `zk-pdb` PodDisruptionBudget, and the `zk` StatefulSet.

```
service "zk-hs" created
service "zk-cs" created
poddisruptionbudget "zk-pdb" created
statefulset "zk" created
```

Use `kubectl get` to watch the StatefulSet controller create the StatefulSet's Pods.

```
kubectl get pods -w -l app=zk
```

Once the `zk-2` Pod is Running and Ready, use `CRTL-C` to terminate kubectl.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0           0s
zk-0      0/1       Pending   0          0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0          19s
zk-0      1/1       Running   0          40s
zk-1      0/1       Pending   0          0s
zk-1      0/1       Pending   0          0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0          18s
zk-1      1/1       Running   0          40s
zk-2      0/1       Pending   0          0s
zk-2      0/1       Pending   0          0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0          19s
zk-2      1/1       Running   0          40s
```

The StatefulSet controller creates three Pods, and each Pod has a container with a ZooKeeper 3.4.9 server.

Facilitating Leader Election

As there is no terminating algorithm for electing a leader in an anonymous network, Zab requires explicit membership configuration in order to perform leader election. Each server in the ensemble needs to have a unique identifier, all servers need to know the global set of identifiers, and each identifier needs to be associated with a network address.

Use `kubectl exec` to get the hostnames of the Pods in the `zk` StatefulSet.

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname; done
```

The StatefulSet controller provides each Pod with a unique hostname based on its ordinal index. The hostnames take the form `<statefulset name>-<ordinal index>`. As the `replicas` field of the `zk` StatefulSet is set to `3`, the Set's controller creates three Pods with their hostnames set to `zk-0`, `zk-1`, and `zk-2`.

```
zk-0
zk-1
zk-2
```

The servers in a ZooKeeper ensemble use natural numbers as unique identifiers, and each server's identifier is stored in a file called `myid` in the server's data directory.

Examine the contents of the `myid` file for each server.

```
for i in 0 1 2; do echo "myid zk-$i";kubectl exec zk-$i -- cat /var/lib/zookeeper/
```

As the identifiers are natural numbers and the ordinal indices are non-negative integers, you can generate an identifier by adding one to the ordinal.

```
myid zk-0
1
myid zk-1
2
myid zk-2
3
```

Get the FQDN (Fully Qualified Domain Name) of each Pod in the `zk` StatefulSet.

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname -f; done
```

The `zk-hs` Service creates a domain for all of the Pods, `zk-headless.default.svc.cluster.local`.

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

The A records in Kubernetes DNS resolve the FQDNs to the Pods' IP addresses. If the Pods are rescheduled, the A records will be updated with the Pods' new IP addresses, but the A record's names will not change.

ZooKeeper stores its application configuration in a file named `zoo.cfg`. Use `kubectl exec` to view the contents of the `zoo.cfg` file in the `zk-0` Pod.

```
kubectl exec zk-0 -- cat /opt/zookeeper/conf/zoo.cfg
```

For the `server.1`, `server.2`, and `server.3` properties at the bottom of the file, the `1`, `2`, and `3` correspond to the identifiers in the ZooKeeper servers' `myid` files. They are set to the FQDNs for the Pods in the `zk` StatefulSet.

```
clientPort=2181
dataDir=/var/lib/zookeeper/data
dataLogDir=/var/lib/zookeeper/log
tickTime=2000
initLimit=10
syncLimit=2000
maxClientCnxns=60
minSessionTimeout= 4000
maxSessionTimeout= 40000
autopurge.snapRetainCount=3
autopurge.purgeInteval=0
server.1=zk-0.zk-headless.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-headless.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-headless.default.svc.cluster.local:2888:3888
```

Achieving Consensus

Consensus protocols require that the identifiers of each participant be unique. No two participants in the Zab protocol should claim the same unique identifier. This is necessary to allow the processes in the system to agree on which processes have committed which data. If two Pods were launched with the same ordinal, two ZooKeeper servers would both identify themselves as the same server.

```
kubectl get pods -w -l app=zk
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

The A records for each Pod are only entered when the Pod becomes Ready. Therefore, the FQDNs of the ZooKeeper servers will only resolve to a single endpoint, and that endpoint will be the unique ZooKeeper server claiming the identity configured in its `myid` file.

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

This ensures that the `servers` properties in the ZooKeepers' `zoo.cfg` files represents a correctly configured ensemble.

```
server.1=zk-0.zk-hs.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-hsdefault.svc.cluster.local:2888:3888
```

When the servers use the Zab protocol to attempt to commit a value, they will either achieve consensus and commit the value (if leader election has succeeded and at least two of the Pods are Running and Ready), or they will fail to do so (if either of the aforementioned conditions are not met). No state will arise where one server acknowledges a write on behalf of another.

Sanity Testing the Ensemble

The most basic sanity test is to write some data to one ZooKeeper server and to read the data from another.

Use the `zkCli.sh` script to write `world` to the path `/hello` on the `zk-0` Pod.

```
kubectl exec zk-0 zkCli.sh create /hello world
```

This will write `world` to the `/hello` path in the ensemble.

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
Created /hello
```

Get the data from the `zk-1` Pod.

```
kubectl exec zk-1 zkCli.sh get /hello
```

The data that you created on `zk-0` is available on all of the servers in the ensemble.

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Providing Durable Storage

As mentioned in the ZooKeeper Basics section, ZooKeeper commits all entries to a durable WAL, and periodically writes snapshots in memory state, to storage media. Using WALs to provide durability is a common technique for applications that use consensus protocols to achieve a replicated state machine and for storage applications in general.

Use `kubectl delete` to delete the `zk` StatefulSet.

```
kubectl delete statefulset zk
statefulset "zk" deleted
```

Watch the termination of the Pods in the StatefulSet.

```
kubectl get pods -w -l app=zk
```

When `zk-0` if fully terminated, use `CRTL-C` to terminate kubectl.

```
zk-2      1/1       Terminating   0         9m
zk-0      1/1       Terminating   0         11m
zk-1      1/1       Terminating   0         10m
zk-2      0/1       Terminating   0         9m
zk-2      0/1       Terminating   0         9m
zk-2      0/1       Terminating   0         9m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
```

Reapply the manifest in `zookeeper.yaml`.

```
kubectl apply -f https://k8s.io/docs/tutorials/stateful-application/zookeeper.yaml
```

The `zk` StatefulSet will be created, but, as they already exist, the other API Objects in the manifest will not be modified.

Watch the StatefulSet controller recreate the StatefulSet's Pods.

```
kubectl get pods -w -l app=zk
```

Once the `zk-2` Pod is Running and Ready, use `CRTL-C` to terminate kubectl.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

Get the value you entered during the sanity test, from the `zk-2` Pod.

```
kubectl exec zk-2 zkCli.sh get /hello
```

Even though all of the Pods in the `zk` StatefulSet have been terminated and recreated, the ensemble still serves the original value.

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

The `volumeClaimTemplates` field, of the `zk` StatefulSet's `spec`, specifies a PersistentVolume that will be provisioned for each Pod.

```
volumeClaimTemplates:
  - metadata:
      name: datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 20Gi
```

The StatefulSet controller generates a PersistentVolumeClaim for each Pod in the StatefulSet.

Get the StatefulSet's PersistentVolumeClaims.

```
kubectl get pvc -l app=zk
```

When the StatefulSet recreated its Pods, the Pods' PersistentVolumes were remounted.

```
NAME           STATUS    VOLUME                                     CAPACITY   ACC
datadir-zk-0   Bound     pvc-bed742cd-bcb1-11e6-994f-42010a800002   20Gi       RWO
datadir-zk-1   Bound     pvc-bedd27d2-bcb1-11e6-994f-42010a800002   20Gi       RWO
datadir-zk-2   Bound     pvc-bee0817e-bcb1-11e6-994f-42010a800002   20Gi       RWO
```

The `volumeMounts` section of the StatefulSet's container `template` causes the PersistentVolumes to be mounted to the ZooKeeper servers' data directories.

```
volumeMounts:
        - name: datadir
          mountPath: /var/lib/zookeeper
```

When a Pod in the `zk` StatefulSet is (re)scheduled, it will always have the same PersistentVolume mounted to the ZooKeeper server's data directory. Even when the Pods are rescheduled, all of the writes made to the ZooKeeper servers' WALs, and all of their snapshots, remain durable.

Ensuring Consistent Configuration

As noted in the Facilitating Leader Election and Achieving Consensus sections, the servers in a ZooKeeper ensemble require consistent configuration in order to elect a leader and form a quorum. They also require consistent configuration of the Zab protocol in order for the protocol to work correctly over a network. In our example we achive consistent configuration by embedding the configuration directly into the manifest.

Get the `zk` StatefulSet.

```
 kubectl get sts zk -o yaml
...
 command:
        - sh
        - -c
        - "start-zookeeper \
          --servers=3 \
          --data_dir=/var/lib/zookeeper/data \
          --data_log_dir=/var/lib/zookeeper/data/log \
          --conf_dir=/opt/zookeeper/conf \
          --client_port=2181 \
          --election_port=3888 \
          --server_port=2888 \
          --tick_time=2000 \
          --init_limit=10 \
          --sync_limit=5 \
          --heap=512M \
          --max_client_cnxns=60 \
          --snap_retain_count=3 \
          --purge_interval=12 \
          --max_session_timeout=40000 \
          --min_session_timeout=4000 \
          --log_level=INFO"
...
```

Notice that the command used to start the ZooKeeper servers passed the configuration as command line parameter. Enviornment variables are another, equally good, way to pass configuration to ensemble.

Configuring Logging

One of the files generated by the `zkGenConfig.sh` script controls ZooKeeper's logging. ZooKeeper uses Log4j, and, by default, it uses a time and size based rolling file appender for its logging configuration. Get the logging configuration from one of Pods in the `zk` StatefulSet.

```
kubectl exec zk-0 cat /usr/etc/zookeeper/log4j.properties
```

The logging configuration below will cause the ZooKeeper process to write all of its logs to the standard output file stream.

```
zookeeper.root.logger=CONSOLE
zookeeper.console.threshold=INFO
log4j.rootLogger=${zookeeper.root.logger}
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=${zookeeper.console.threshold}
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] - %-5p
```

This is the simplest possible way to safely log inside the container. As the application's logs are being written to standard out, Kubernetes will handle log rotation for you. Kubernetes also implements a sane retention policy that ensures application logs written to standard out and standard error do not exhaust local storage media.

Use `kubectl logs` to retrieve the last few log lines from one of the Pods.

```
kubectl logs zk-0 --tail 20
```

Application logs that are written to standard out or standard error are viewable using `kubectl logs` and from the Kubernetes Dashboard.

```
2016-12-06 19:34:16,236 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:16,237 [myid:1] - INFO  [Thread-1136:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:26,156 [myid:1] - INFO  [Thread-1137:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:26,226 [myid:1] - INFO  [Thread-1138:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:36,151 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:36,152 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:36,152 [myid:1] - INFO  [Thread-1139:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:36,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:36,231 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:36,231 [myid:1] - INFO  [Thread-1140:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:46,149 [myid:1] - INFO  [Thread-1141:NIOServerCnxn@1008] - Closed
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:218
2016-12-06 19:34:46,230 [myid:1] - INFO  [Thread-1142:NIOServerCnxn@1008] - Closed
```

Kubernetes also supports more powerful, but more complex, logging integrations with Logging Using Stackdriver and Logging Using Elasticsearch and Kibana. For cluster level log shipping and aggregation, you should consider deploying a sidecar container to rotate and ship your logs.

Configuring a Non-Privileged User

The best practices with respect to allowing an application to run as a privileged user inside of a container are a matter of debate. If your organization requires that applications be run as a non-privileged user you can use a SecurityContext to control the user that the entry point runs as.

The `zk` StatefulSet's Pod `template` contains a SecurityContext.

```
securityContext:
  runAsUser: 1000
  fsGroup: 1000
```

In the Pods' containers, UID 1000 corresponds to the zookeeper user and GID 1000 corresponds to the zookeeper group.

Get the ZooKeeper process information from the `zk-0` Pod.

```
kubectl exec zk-0 -- ps -elf
```

As the `runAsUser` field of the `securityContext` object is set to 1000, instead of running as root, the ZooKeeper process runs as the zookeeper user.

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S zookeep+     1     0  0  80   0 -  1127 -      20:46 ?        00:00:00 sh -c z
0 S zookeep+    27     1  0  80   0 - 1155556 -    20:46 ?        00:00:19 /usr/li
```

By default, when the Pod's PersistentVolume is mounted to the ZooKeeper server's data directory, it is only accessible by the root user. This configuration prevents the ZooKeeper process from writing to its WAL and storing its snapshots.

Get the file permissions of the ZooKeeper data directory on the `zk-0` Pod.

```
kubectl exec -ti zk-0 -- ls -ld /var/lib/zookeeper/data
```

As the `fsGroup` field of the `securityContext` object is set to 1000, the ownership of the Pods' PersistentVolumes is set to the zookeeper group, and the ZooKeeper process is able to successfully read and write its data.

```
drwxr-sr-x 3 zookeeper zookeeper 4096 Dec  5 20:45 /var/lib/zookeeper/data
```

Managing the ZooKeeper Process

The ZooKeeper documentation indicates that "You will want to have a supervisory process that manages each of your ZooKeeper server processes (JVM)." Utilizing a watchdog (supervisory process) to restart failed processes in a distributed system is a common pattern. When deploying an application in Kubernetes, rather than using an external utility as a supervisory process, you should use Kubernetes as the watchdog for your application.

Updating the Ensemble

The `zk` StatefulSet is configured to use the RollingUpdate update strategy.

You can use `kubectl patch` to update the number of `cpus` allocated to the servers.

```
kubectl patch sts zk --type='json' -p='[{"op": "replace", "path": "/spec/template/
statefulset "zk" patched
```

Use `kubectl rollout status` to watch the status of the update.

```
kubectl rollout status sts/zk
waiting for statefulset rolling update to complete 0 pods at revision zk-5db449966
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 1 pods at revision zk-5db449966
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 2 pods at revision zk-5db449966
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
statefulset rolling update complete 3 pods at revision zk-5db4499664...
```

The Pods are terminated, one at a time, in reverse ordinal order, and they are recreated with the new configuration. This ensures that quorum is maintained during a rolling update.

Use `kubectl rollout history` to view a history or previous configurations.

```
kubectl rollout history sts/zk
statefulsets "zk"
REVISION
1
2
```

Use `kubectl rollout undo` to roll back the modification.

```
kubectl rollout undo sts/zk
statefulset "zk" rolled back
```

Handling Process Failure

Restart Policies control how Kubernetes handles process failures for the entry point of the container in a Pod. For Pods in a StatefulSet, the only appropriate RestartPolicy is Always, and this is the default value. For stateful applications you should **never** override the default policy.

Examine the process tree for the ZooKeeper server running in the `zk-0` Pod.

```
kubectl exec zk-0 -- ps -ef
```

The command used as the container's entry point has PID 1, and the ZooKeeper process, a child of the entry point, has PID 23.

```
UID         PID  PPID  C STIME TTY          TIME CMD
zookeep+      1     0  0 15:03 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.s
zookeep+     27     1  0 15:03 ?        00:00:03 /usr/lib/jvm/java-8-openjdk-amd64/
```

In one terminal watch the Pods in the `zk` StatefulSet.

```
kubectl get pod -w -l app=zk
```

In another terminal, kill the ZooKeeper process in Pod `zk-0`.

```
 kubectl exec zk-0 -- pkill java
```

The death of the ZooKeeper process caused its parent process to terminate. As the RestartPolicy of the container is Always, the parent process was relaunched.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   0          21m
zk-1      1/1       Running   0          20m
zk-2      1/1       Running   0          19m
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Error     0          29m
zk-0      0/1       Running   1         29m
zk-0      1/1       Running   1         29m
```

If your application uses a script (such as zkServer.sh) to launch the process that implements the application's business logic, the script must terminate with the child process. This ensures that Kubernetes will restart the application's container when the process implementing the application's business logic fails.

Testing for Liveness

Configuring your application to restart failed processes is not sufficient to keep a distributed system healthy. There are many scenarios where a system's processes can be both alive and unresponsive, or otherwise unhealthy. You should use liveness probes in order to notify Kubernetes that your application's processes are unhealthy and should be restarted.

The Pod `template` for the `zk` StatefulSet specifies a liveness probe.

```
 livenessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
```

The probe calls a simple bash script that uses the ZooKeeper `ruok` four letter word to test the server's health.

```
ZK_CLIENT_PORT=${ZK_CLIENT_PORT:-2181}
OK=$(echo ruok | nc 127.0.0.1 $ZK_CLIENT_PORT)
if [ "$OK" == "imok" ]; then
    exit 0
else
    exit 1
fi
```

In one terminal window, watch the Pods in the `zk` StatefulSet.

```
kubectl get pod -w -l app=zk
```

In another window, delete the `zkOk.sh` script from the file system of Pod `zk-0`.

```
kubectl exec zk-0 -- rm /opt/zookeeper/bin/zkOk.sh
```

When the liveness probe for the ZooKeeper process fails, Kubernetes will automatically restart the process for you, ensuring that unhealthy processes in the ensemble are restarted.

```
kubectl get pod -w -l app=zk
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   0          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Running   0          1h
zk-0      0/1       Running   1         1h
zk-0      1/1       Running   1         1h
```

Testing for Readiness

Readiness is not the same as liveness. If a process is alive, it is scheduled and healthy. If a process is ready, it is able to process input. Liveness is a necessary, but not sufficient, condition for readiness. There are many cases, particularly during initialization and termination, when a process can be alive but not ready.

If you specify a readiness probe, Kubernetes will ensure that your application's processes will not receive network traffic until their readiness checks pass.

For a ZooKeeper server, liveness implies readiness. Therefore, the readiness probe from the `zookeeper.yaml` manifest is identical to the liveness probe.

```
  readinessProbe:
          exec:
            command:
            - "zkOk.sh"
          initialDelaySeconds: 15
          timeoutSeconds: 5
```

Even though the liveness and readiness probes are identical, it is important to specify both. This ensures that only healthy servers in the ZooKeeper ensemble receive network traffic.

Tolerating Node Failure

ZooKeeper needs a quorum of servers in order to successfully commit mutations to data. For a three server ensemble, two servers must be healthy in order for writes to succeed. In quorum based systems, members are deployed across failure domains to ensure availability. In order to avoid an outage, due to the loss of an individual machine, best practices preclude co-locating multiple instances of the application on the same machine.

By default, Kubernetes may co-locate Pods in a StatefulSet on the same node. For the three server ensemble you created, if two servers reside on the same node, and that node fails, the clients of your ZooKeeper service will experience an outage until at least one of the Pods can be rescheduled.

You should always provision additional capacity to allow the processes of critical systems to be rescheduled in the event of node failures. If you do so, then the outage will only last until the Kubernetes scheduler reschedules one of the ZooKeeper servers. However, if you want your service to tolerate node failures with no downtime, you should set `podAntiAffinity`.

Get the nodes for Pods in the `zk` Stateful Set.

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo ""; d
```

All of the Pods in the `zk` StatefulSet are deployed on different nodes.

```
kubernetes-minion-group-cxpk
kubernetes-minion-group-a5aq
kubernetes-minion-group-2g2d
```

This is because the Pods in the `zk` StatefulSet have a PodAntiAffinity specified.

```
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - zk-headless
              topologyKey: "kubernetes.io/hostname"
```

The `requiredDuringSchedulingIgnoredDuringExecution` field tells the Kubernetes Scheduler that it should never co-locate two Pods from the `zk-headless` Service in the domain defined by the `topologyKey`. The `topologyKey` `kubernetes.io/hostname` indicates that the domain is an individual node. Using different rules, labels, and selectors, you can extend this technique to spread your ensemble across physical, network, and power failure domains.

Surviving Maintenance

In this section you will cordon and drain nodes. If you are using this tutorial on a shared cluster, be sure that this will not adversely affect other tenants.

The previous section showed you how to spread your Pods across nodes to survive unplanned node failures, but you also need to plan for temporary node failures that occur due to planned maintenance.

Get the nodes in your cluster.

```
kubectl get nodes
```

Use `kubectl cordon` to cordon all but four of the nodes in your cluster.

```
kubectl cordon < node name >
```

Get the `zk-pdb` PodDisruptionBudget.

```
kubectl get pdb zk-pdb
```

The `max-unavailable` field indicates to Kubernetes that at most one Pod from `zk` StatefulSet can be unavailable at any time.

```
NAME      MIN-AVAILABLE   MAX-UNAVAILABLE   ALLOWED-DISRUPTIONS   AGE
zk-pdb    N/A             1                 1
```

In one terminal, watch the Pods in the `zk` StatefulSet.

```
kubectl get pods -w -l app=zk
```

In another terminal, get the nodes that the Pods are currently scheduled on.

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo ""; d
kubernetes-minion-group-pb41
kubernetes-minion-group-ixsl
kubernetes-minion-group-i4c4
```

Use `kubectl drain` to cordon and drain the node on which the `zk-0` Pod is scheduled.

```
kubectl drain $(kubectl get pod zk-0 --template {{.spec.nodeName}}) --ignore-daemo
node "kubernetes-minion-group-pb41" cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or D
pod "zk-0" deleted
node "kubernetes-minion-group-pb41" drained
```

As there are four nodes in your cluster, `kubectl drain`, succeeds and the `zk-0` is rescheduled to another node.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
```

Keep watching the StatefulSet's Pods in the first terminal and drain the node on which `zk-1` is scheduled.

```
kubectl drain $(kubectl get pod zk-1 --template {{.spec.nodeName}}) --ignore-daemo
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or D
pod "zk-1" deleted
node "kubernetes-minion-group-ixsl" drained
```

The `zk-1` Pod can not be scheduled. As the `zk` StatefulSet contains a PodAntiAffinity rule preventing co-location of the Pods, and as only two nodes are schedulable, the Pod will remain in a Pending state.

```
kubectl get pods -w -l app=zk
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
zk-1      1/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
```

Continue to watch the Pods of the stateful set, and drain the node on which `zk-2` is scheduled.

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemo
node "kubernetes-minion-group-i4c4" cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or D
WARNING: Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog; Deleti
There are pending pods when an error occurred: Cannot evict pod as it would violat
pod/zk-2
```

Use `CRTL-C` to terminate to kubectl.

You can not drain the third node because evicting `zk-2` would violate `zk-budget`. However, the node will remain cordoned.

Use `zkCli.sh` to retrieve the value you entered during the sanity test from `zk-0`.

```
kubectl exec zk-0 zkCli.sh get /hello
```

The service is still available because its PodDisruptionBudget is respected.

```
WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x200000002
ctime = Wed Dec 07 00:08:59 UTC 2016
mZxid = 0x200000002
mtime = Wed Dec 07 00:08:59 UTC 2016
pZxid = 0x200000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Use `kubectl uncordon` to uncordon the first node.

```
kubectl uncordon kubernetes-minion-group-pb41
node "kubernetes-minion-group-pb41" uncordoned
```

`zk-1` is rescheduled on this node. Wait until `zk-1` is Running and Ready.

```
kubectl get pods -w -l app=zk
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2           2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
zk-1      1/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         12m
zk-1      0/1       ContainerCreating   0         12m
zk-1      0/1       Running   0         13m
zk-1      1/1       Running   0         13m
```

Attempt to drain the node on which `zk-2` is scheduled.

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemo
node "kubernetes-minion-group-i4c4" already cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or D
pod "heapster-v1.2.0-2604621511-wht1r" deleted
pod "zk-2" deleted
node "kubernetes-minion-group-i4c4" drained
```

This time `kubectl drain` succeeds.

Uncordon the second node to allow `zk-2` to be rescheduled.

```
kubectl uncordon kubernetes-minion-group-ixsl
node "kubernetes-minion-group-ixsl" uncordoned
```

You can use `kubectl drain` in conjunction with PodDisruptionBudgets to ensure that your service remains available during maintenance. If drain is used to cordon nodes and evict pods prior to taking the node offline for maintenance, services that express a disruption budget will have that budget respected. You should always allocate additional capacity for critical services so that their Pods can be immediately rescheduled.

Cleaning up

- Use `kubectl uncordon` to uncordon all the nodes in your cluster.
- You will need to delete the persistent storage media for the PersistentVolumes used in this tutorial. Follow the necessary steps, based on your environment, storage configuration, and provisioning method, to ensure that all storage is reclaimed.

AppArmor

FEATURE STATE: `Kubernetes v1.4` beta

AppArmor is a Linux kernel security module that supplements the standard Linux user and group based permissions to confine programs to a limited set of resources. AppArmor can be configured for any application to reduce its potential attack surface and provide greater in-depth defense. It is configured through profiles tuned to whitelist the access needed by a specific program or container, such as Linux capabilities, network access, file permissions, etc. Each profile can be run in either *enforcing* mode, which blocks access to disallowed resources, or *complain* mode, which only reports violations.

AppArmor can help you to run a more secure deployment by restricting what containers are allowed to do, and/or provide better auditing through system logs. However, it is important to keep in mind that AppArmor is not a silver bullet and can only do so much to protect against exploits in your application code. It is important to provide good, restrictive profiles, and harden your applications and cluster from other angles as well.

Objectives

- See an example of how to load a profile on a node
- Learn how to enforce the profile on a Pod
- Learn how to check that the profile is loaded
- See what happens when a profile is violated
- See what happens when a profile cannot be loaded

Before you begin

Make sure:

1. Kubernetes version is at least v1.4 – Kubernetes support for AppArmor was added in v1.4. Kubernetes components older than v1.4 are not aware of the new AppArmor annotations, and will **silently ignore** any AppArmor settings that are provided. To ensure that your Pods are receiving the expected protections, it is important to verify the Kubelet version of your nodes:

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.statu
gke-test-default-pool-239f5d02-gyn2: v1.4.0
gke-test-default-pool-239f5d02-x1kf: v1.4.0
gke-test-default-pool-239f5d02-xwux: v1.4.0
```

2. AppArmor kernel module is enabled – For the Linux kernel to enforce an AppArmor profile, the AppArmor kernel module must be installed and enabled. Several distributions enable the module by default, such as Ubuntu and SUSE, and many others provide optional support. To check whether the module is enabled, check the `/sys/module/apparmor/parameters/enabled` file:

```
$ cat /sys/module/apparmor/parameters/enabled
Y
```

If the Kubelet contains AppArmor support (>= v1.4), it will refuse to run a Pod with AppArmor options if the kernel module is not enabled.

Note: Ubuntu carries many AppArmor patches that have not been merged into the upstream Linux kernel, including patches that add additional hooks and features. Kubernetes has only been tested with the upstream version, and does not promise support for other features.

3. Container runtime is Docker – Currently the only Kubernetes-supported container runtime that also supports AppArmor is Docker. As more runtimes add AppArmor support, the options will be expanded. You can verify that your nodes are running docker with:

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.statu
gke-test-default-pool-239f5d02-gyn2: docker://1.11.2
gke-test-default-pool-239f5d02-x1kf: docker://1.11.2
gke-test-default-pool-239f5d02-xwux: docker://1.11.2
```

If the Kubelet contains AppArmor support (>= v1.4), it will refuse to run a Pod with AppArmor options if the runtime is not Docker.

4. Profile is loaded – AppArmor is applied to a Pod by specifying an AppArmor profile that each container should be run with. If any of the specified profiles is not already loaded in the kernel, the Kubelet (>= v1.4) will reject the Pod. You can view which profiles are loaded on a node by checking the `/sys/kernel/security/apparmor/profiles` file. For example:

```
$ ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/security/appar
apparmor-test-deny-write (enforce)
apparmor-test-audit-write (enforce)
docker-default (enforce)
k8s-nginx (enforce)
```

For more details on loading profiles on nodes, see Setting up nodes with profiles.

As long as the Kubelet version includes AppArmor support (>= v1.4), the Kubelet will reject a Pod with AppArmor options if any of the prerequisites are not met. You can also verify AppArmor support on nodes by checking the node ready condition message (though this is likely to be removed in a later release):

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {.status.con
gke-test-default-pool-239f5d02-gyn2: kubelet is posting ready status. AppArmor ena
gke-test-default-pool-239f5d02-x1kf: kubelet is posting ready status. AppArmor ena
gke-test-default-pool-239f5d02-xwux: kubelet is posting ready status. AppArmor ena
```

Securing a Pod

Note: AppArmor is currently in beta, so options are specified as annotations. Once support graduates to general availability, the annotations will be replaced with first-class fields (more details in Upgrade path to GA).

AppArmor profiles are specified *per-container*. To specify the AppArmor profile to run a Pod container with, add an annotation to the Pod's metadata:

```
container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_ref>
```

Where `<container_name>` is the name of the container to apply the profile to, and `<profile_ref>` specifies the profile to apply. The `profile_ref` can be one of:

- `runtime/default` to apply the runtime's default profile
- `localhost/<profile_name>` to apply the profile loaded on the host with the name `<profile_name>`

See the API Reference for the full details on the annotation and profile name formats.

Kubernetes AppArmor enforcement works by first checking that all the prerequisites have been met, and then forwarding the profile selection to the container runtime for enforcement. If the prerequisites have not been met, the Pod will be rejected, and will not run.

To verify that the profile was applied, you can look for the AppArmor security option listed in the container created event:

```
$ kubectl get events | grep Created
22s        22s         1         hello-apparmor     Pod       spec.containers{hell
```

You can also verify directly that the container's root process is running with the correct profile by checking its proc attr:

```
$ kubectl exec <pod_name> cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

Example

This example assumes you have already set up a cluster with AppArmor support.

First, we need to load the profile we want to use onto our nodes. The profile we'll use simply denies all file writes:

deny-write.profile

```
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>

  file,

  # Deny all file writes.
  deny /** w,
}
```

Since we don't know where the Pod will be scheduled, we'll need to load the profile on all our nodes. For this example we'll just use SSH to install the profiles, but other approaches are discussed in Setting up nodes with profiles.

```
$ NODES=(
    # The SSH-accessible domain names of your nodes
    gke-test-default-pool-239f5d02-gyn2.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-x1kf.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-xwux.us-central1-a.my-k8s)
$ for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_parser -q <<EOF
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>

  file,

  # Deny all file writes.
  deny /** w,
}
EOF'
done
```

Next, we'll run a simple "Hello AppArmor" pod with the deny-write profile:

hello-apparmor-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    # Tell Kubernetes to apply the AppArmor profile "k8s-apparmor-example-deny-wri
    # Note that this is ignored if the Kubernetes node is not running version 1.4
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-e
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

```
$ kubectl create -f ./hello-apparmor-pod.yaml
```

If we look at the pod events, we can see that the Pod container was created with the AppArmor profile "k8s-apparmor-example-deny-write":

```
$ kubectl get events | grep hello-apparmor
14s         14s          1          hello-apparmor   Pod
14s         14s          1          hello-apparmor   Pod        spec.containers{hello}
13s         13s          1          hello-apparmor   Pod        spec.containers{hello}
13s         13s          1          hello-apparmor   Pod        spec.containers{hello}
13s         13s          1          hello-apparmor   Pod        spec.containers{hello}
```

We can verify that the container is actually running with that profile by checking its proc attr:

```
$ kubectl exec hello-apparmor cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

Finally, we can see what happens if we try to violate the profile by writing to a file:

```
$ kubectl exec hello-apparmor touch /tmp/test
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero exit code:
```

To wrap up, let's look at what happens if we try to specify a profile that hasn't been loaded:

```
$ kubectl create -f /dev/stdin <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor-2
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-e
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
EOF
pod "hello-apparmor-2" created

$ kubectl describe pod hello-apparmor-2
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kubernetes.io/hello=localhost/k8s-
Status:        Pending
```

```
Reason:         AppArmor
Message:        Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-wr
IP:
Controllers:    <none>
Containers:
  hello:
    Container ID:
    Image:      busybox
    Image ID:
    Port:
    Command:
      sh
      -c
      echo 'Hello AppArmor!' && sleep 1h
    State:              Waiting
      Reason:           Blocked
    Ready:              False
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-dnz7v (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-dnz7v:
    Type:    Secret (a volume populated by a Secret)
    SecretName:    default-token-dnz7v
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen    LastSeen    Count    From                        SubobjectPath    T
  ---------    --------    -----    ----                        -------------    -
  23s          23s         1        {default-scheduler }                         N
  23s          23s         1        {kubelet e2e-test-stclair-minion-group-t1f5}
```

Note the pod status is Failed, with a helpful error message:

```
Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" is not
loaded
```

. An event was also recorded with the same message.

Administration

Setting up nodes with profiles

Kubernetes does not currently provide any native mechanisms for loading AppArmor profiles onto nodes. There are lots of ways to setup the profiles though, such as:

- Through a DaemonSet that runs a Pod on each node to ensure the correct profiles are loaded. An example implementation can be found here.
- At node initialization time, using your node initialization scripts (e.g. Salt, Ansible, etc.) or image.
- By copying the profiles to each node and loading them through SSH, as demonstrated in the Example.

The scheduler is not aware of which profiles are loaded onto which node, so the full set of profiles must be loaded onto every node. An alternative approach is to add a node label for each profile (or class of profiles) on the node, and use a node selector to ensure the Pod is run on a node with the required profile.

Restricting profiles with the PodSecurityPolicy

If the PodSecurityPolicy extension is enabled, cluster-wide AppArmor restrictions can be applied. To enable the PodSecurityPolicy, two flags must be set on the `apiserver`:

```
--admission-control=PodSecurityPolicy[,others...]
--runtime-config=extensions/v1beta1/podsecuritypolicy[,others...]
```

With the extension enabled, the AppArmor options can be specified as annotations on the PodSecurityPolicy:

```
apparmor.security.beta.kubernetes.io/defaultProfileName: <profile_ref>
apparmor.security.beta.kubernetes.io/allowedProfileNames: <profile_ref>[,others...
```

The default profile name option specifies the profile to apply to containers by default when none is specified. The allowed profile names option specifies a list of profiles that Pod containers are allowed to be run with. If both options are provided, the default must be allowed. The profiles are specified in the same format as on containers. See the API Reference for the full specification.

Disabling AppArmor

If you do not want AppArmor to be available on your cluster, it can be disabled by a command-line flag:

```
--feature-gates=AppArmor=false
```

When disabled, any Pod that includes an AppArmor profile will fail validation with a “Forbidden” error. Note that by default docker always enables the “docker-default” profile on non-privileged pods (if the AppArmor kernel module is enabled), and will continue to do so even if the feature-gate is disabled. The option to disable AppArmor will be removed when AppArmor graduates to general availability (GA).

Upgrading to Kubernetes v1.4 with AppArmor

No action is required with respect to AppArmor to upgrade your cluster to v1.4. However, if any existing pods had an AppArmor annotation, they will not go through validation (or PodSecurityPolicy admission). If permissive profiles are loaded on the nodes, a malicious user could pre-apply a permissive profile to escalate the pod privileges above the docker-default. If this is a concern, it is recommended to scrub the cluster of any pods containing an annotation with `apparmor.security.beta.kubernetes.io`.

Upgrade path to General Availability

When AppArmor is ready to be graduated to general availability (GA), the options currently specified through annotations will be converted to fields. Supporting all the upgrade and downgrade paths through the transition is very nuanced, and will be explained in detail when the transition occurs. We will commit to supporting both fields and annotations for at least 2 releases, and will explicitly reject the annotations for at least 2 releases after that.

Authoring Profiles

Getting AppArmor profiles specified correctly can be a tricky business. Fortunately there are some tools to help with that:

- `aa-genprof` and `aa-logprof` generate profile rules by monitoring an application's activity and logs, and admitting the actions it takes. Further instructions are provided by the AppArmor documentation.
- bane is an AppArmor profile generator for Docker that uses a simplified profile language.

It is recommended to run your application through Docker on a development workstation to generate the profiles, but there is nothing preventing running the tools on the Kubernetes node where your Pod is running.

To debug problems with AppArmor, you can check the system logs to see what, specifically, was denied. AppArmor logs verbose messages to `dmesg`, and errors can usually be found in the system logs or through `journalctl`. More information is provided in AppArmor failures.

API Reference

Pod Annotation

Specifying the profile a container will run with:

- **key**: `container.apparmor.security.beta.kubernetes.io/<container_name>` Where `<container_name>` matches the name of a container in the Pod. A separate profile can be specified for each container in the Pod.
- **value**: a profile reference, described below

Profile Reference

- `runtime/default`: Refers to the default runtime profile.
  - Equivalent to not specifying a profile (without a PodSecurityPolicy default), except it still requires AppArmor to be enabled.
  - For Docker, this resolves to the `docker-default` profile for non-privileged containers, and unconfined (no profile) for privileged containers.
- `localhost/<profile_name>`: Refers to a profile loaded on the node (localhost) by name.

  - The possible profile names are detailed in the core policy reference.

Any other profile reference format is invalid.

PodSecurityPolicy Annotations

Specifying the default profile to apply to containers when none is provided:

- **key**: `apparmor.security.beta.kubernetes.io/defaultProfileName`
- **value**: a profile reference, described above

Specifying the list of profiles Pod containers is allowed to specify:

- **key**: `apparmor.security.beta.kubernetes.io/allowedProfileNames`
- **value**: a comma-separated list of profile references (described above)
  - Although an escaped comma is a legal character in a profile name, it cannot be explicitly allowed here.

What's next

Additional resources:

- Quick guide to the AppArmor profile language
- AppArmor core policy reference

Using Source IP

Applications running in a Kubernetes cluster find and communicate with each other, and the outside world, through the Service abstraction. This document explains what happens to the source IP of packets sent to different types of Services, and how you can toggle this behavior according to your needs.

Objectives

- Expose a simple application through various types of Services
- Understand how each Service type handles source IP NAT
- Understand the tradeoffs involved in preserving source IP

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube, or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Terminology

This document makes use of the following terms:

- NAT: network address translation
- Source NAT: replacing the source IP on a packet, usually with a node's IP
- Destination NAT: replacing the destination IP on a packet, usually with a pod IP
- VIP: a virtual IP, such as the one assigned to every Kubernetes Service
- Kube-proxy: a network daemon that orchestrates Service VIP management on every node

Prerequisites

You must have a working Kubernetes 1.5 cluster to run the examples in this document. The examples use a small nginx webserver that echoes back the source IP of requests it receives through an HTTP header. You can create it as follows:

```
$ kubectl run source-ip-app --image=gcr.io/google_containers/echoserver:1.4
deployment "source-ip-app" created
```

Source IP for Services with Type=ClusterIP

Packets sent to ClusterIP from within the cluster are never source NAT'd if you're running kube-proxy in iptables mode, which is the default since Kubernetes 1.2. Kube-proxy exposes its mode through a `proxyMode` endpoint:

```
$ kubectl get nodes
NAME                           STATUS     AGE     VERSION
kubernetes-minion-group-6jst   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-cx31   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-jj1t   Ready      2h      v1.6.0+fff5156

kubernetes-minion-group-6jst $ curl localhost:10249/proxyMode
iptables
```

You can test source IP preservation by creating a Service over the source IP app:

```
$ kubectl expose deployment source-ip-app --name=clusterip --port=80 --target-port
service "clusterip" exposed

$ kubectl get svc clusterip
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
clusterip    10.0.170.92   <none>        80/TCP    51s
```

And hitting the `ClusterIP` from a pod in the same cluster:

```
$ kubectl run busybox -it --image=busybox --restart=Never --rm
Waiting for pod default/busybox to be running, status is Pending, pod ready: false
If you don't see a command prompt, try pressing enter.

# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.8/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::188a:84ff:feb0:26a5/64 scope link
       valid_lft forever preferred_lft forever

# wget -qO - 10.0.170.92
CLIENT VALUES:
client_address=10.244.3.8
command=GET
...
```

If the client pod and server pod are in the same node, the client_address is the client pod's IP address. However, if the client pod and server pod are in different nodes, the client_address is the client pod's node flannel IP address.

Source IP for Services with Type=NodePort

As of Kubernetes 1.5, packets sent to Services with Type=NodePort are source NAT'd by default. You can test this by creating a `NodePort` Service:

```
$ kubectl expose deployment source-ip-app --name=nodeport --port=80 --target-port=
service "nodeport" exposed

$ NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].nodePort}" services nodeport
$ NODES=$(kubectl get nodes -o jsonpath='{ $.items[*].status.addresses[?(@.type=="
```

If you're running on a cloudprovider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

```
$ for node in $NODES; do curl -s $node:$NODEPORT | grep -i client_address; done
client_address=10.180.1.1
client_address=10.240.0.5
client_address=10.240.0.3
```

Note that these are not the correct client IPs, they're cluster internal IPs. This is what happens:

- Client sends packet to `node2:nodePort`
- `node2` replaces the source IP address (SNAT) in the packet with its own IP address
- `node2` replaces the destination IP on the packet with the pod IP
- packet is routed to node 1, and then to the endpoint
- the pod's reply is routed back to node2
- the pod's reply is sent back to the client

Visually:

```
          client
             \ ^
              \ \
               v \
   node 1 <--- node 2
    | ^   SNAT
    | |   --->
    v |
 endpoint
```

To avoid this, Kubernetes has a feature to preserve the client source IP (check here for feature availability). Setting `service.spec.externalTrafficPolicy` to the value `Local` will only proxy requests to local endpoints, never forwarding traffic to other nodes and thereby preserving the original source IP address. If there are no local endpoints, packets sent to the node are dropped, so you can rely on the correct source-ip in any packet processing rules you might apply a packet that make it through to the endpoint.

Set the `service.spec.externalTrafficPolicy` field as follows:

```
$ kubectl patch svc nodeport -p '{"spec":{"externalTrafficPolicy":"Local"}}'
service "nodeport" patched
```

Now, re-run the test:

```
$ for node in $NODES; do curl --connect-timeout 1 -s $node:$NODEPORT | grep -i cli
client_address=104.132.1.79
```

Note that you only got one reply, with the *right* client IP, from the one node on which the endpoint pod is running.

This is what happens:

- client sends packet to `node2:nodePort`, which doesn't have any endpoints
- packet is dropped
- client sends packet to `node1:nodePort`, which *does* have endpoints
- node1 routes packet to endpoint with the correct source IP

Visually:

```
          client
        ^ /   \
       / /     \
      / v       X
    node 1     node 2
     ^ |
     | |
     | v
  endpoint
```

Source IP for Services with Type=LoadBalancer

As of Kubernetes 1.5, packets sent to Services with Type=LoadBalancer are source NAT'd by default, because all schedulable Kubernetes nodes in the `Ready` state are eligible for loadbalanced traffic. So if packets arrive at a node without an endpoint, the system proxies it to a node *with* an endpoint, replacing the source IP on the packet with the IP of the node (as described in the previous section).

You can test this by exposing the source-ip-app through a loadbalancer

```
$ kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-p
service "loadbalancer" exposed

$ kubectl get svc loadbalancer
NAME           CLUSTER-IP    EXTERNAL-IP       PORT(S)   AGE
loadbalancer   10.0.65.118   104.198.149.140   80/TCP    5m

$ curl 104.198.149.140
CLIENT VALUES:
client_address=10.240.0.5
...
```

However, if you're running on GKE/GCE, setting the same `service.spec.externalTrafficPolicy` field to `Local` forces nodes *without* Service endpoints to remove themselves from the list of nodes eligible for loadbalanced traffic by deliberately failing health checks.

Visually:

```
                      client
                        |
                      lb VIP
                     / ^
                    v /
health check --->   node 1   node 2 <--- health check
        200  <---   ^ |             ---> 500
                    | V
                 endpoint
```

You can test this by setting the annotation:

```
$ kubectl patch svc loadbalancer -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

You should immediately see the `service.spec.healthCheckNodePort` field allocated by Kubernetes:

```
$ kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort
  healthCheckNodePort: 32122
```

The `service.spec.healthCheckNodePort` field points to a port on every node serving the health check at `/healthz`. You can test this:

```
$ kubectl get pod -o wide -l run=source-ip-app
NAME                            READY     STATUS    RESTARTS   AGE       IP
source-ip-app-826191075-qehz4   1/1       Running   0          20h       10.180.1.

kubernetes-minion-group-6jst $ curl localhost:32122/healthz
1 Service Endpoints found

kubernetes-minion-group-jj1t $ curl localhost:32122/healthz
No Service Endpoints Found
```

A service controller running on the master is responsible for allocating the cloud loadbalancer, and when it does so, it also allocates HTTP health checks pointing to this port/path on each node. Wait about 10 seconds for the 2 nodes without endpoints to fail health checks, then curl the lb ip:

```
$ curl 104.198.149.140
CLIENT VALUES:
client_address=104.132.1.79
...
```

Cross platform support

As of Kubernetes 1.5, support for source IP preservation through Services with Type=LoadBalancer is only implemented in a subset of cloudproviders (GCP and Azure). The cloudprovider you're running on might fulfill the request for a loadbalancer in a few different ways:

1. With a proxy that terminates the client connection and opens a new connection to your nodes/endpoints. In such cases the source IP will always be that of the cloud LB, not that of the client.
2. With a packet forwarder, such that requests from the client sent to the loadbalancer VIP end up at the node with the source IP of the client, not an intermediate proxy.

Loadbalancers in the first category must use an agreed upon protocol between the loadbalancer and backend to communicate the true client IP such as the HTTP X-FORWARDED-FOR header, or the proxy protocol. Loadbalancers in the second category can leverage the feature described above by simply creating an HTTP health check pointing at the port stored in the `service.spec.healthCheckNodePort` field on the Service.

Cleaning up

Delete the Services:

```
$ kubectl delete svc -l run=source-ip-app
```

Delete the Deployment, ReplicaSet and Pod:

```
$ kubectl delete deployment source-ip-app
```

What's next

- Learn more about connecting applications via services
- Learn more about loadbalancing