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.

If you would like to write a task page, see Creating a Documentation Pull Request.

Install Tools

Set up Kubernetes tools on your computer.

Administer a Cluster

Learn common tasks for administering a cluster.

Configure Pods and Containers

Perform common configuration tasks for Pods and containers.

Manage Kubernetes Objects

Declarative and imperative paradigms for interacting with the Kubernetes API.

Managing Secrets

Managing confidential settings data using Secrets.

Inject Data Into Applications

Specify configuration and other data for the Pods that run your workload.

Run Applications

Run and manage both stateless and stateful applications.

Run Jobs

Run Jobs using parallel processing.

Access 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

Set up monitoring and logging to troubleshoot a cluster, or debug a containerized application.

Extend Kubernetes

Understand advanced ways to adapt your Kubernetes cluster to the needs of your work environment.

TLS

Understand how to protect traffic within your cluster using Transport Layer Security (TLS).

Manage Cluster Daemons

Perform common tasks for managing a DaemonSet, such as performing a rolling update.

Service Catalog

Install the Service Catalog extension API.

Networking

Learn how to configure networking for your cluster.

Configure a kubelet image credential provider

Configure the kubelet's image credential provider plugin

Extend kubectl with plugins

Extend kubectl by creating and installing kubectl plugins.

Manage HugePages

Configure and manage huge pages as a schedulable resource in a cluster.

Schedule GPUs

Configure and schedule GPUs for use as a resource by nodes in a cluster.

Install Tools

Set up Kubernetes tools on your computer.

kubectl

The Kubernetes command-line tool, kubectl, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the `kubectl` reference documentation.

kubectl is installable on a variety of Linux platforms, macOS and Windows. Find your preferred operating system below.

- Install kubectl on Linux
- Install kubectl on macOS
- Install kubectl on Windows

kind

`kind` lets you run Kubernetes on your local computer. This tool requires that you have Docker installed and configured.

The kind Quick Start page shows you what you need to do to get up and running with kind.

View kind Quick Start Guide

minikube

Like `kind`, `minikube` is a tool that lets you run Kubernetes locally. `minikube` runs a single-node Kubernetes cluster on your personal computer (including Windows, macOS and Linux PCs) so that you can try out Kubernetes, or for daily development work.

You can follow the official Get Started! guide if your focus is on getting the tool installed.

View minikube Get Started! Guide

Once you have `minikube` working, you can use it to run a sample application.

kubeadm

You can use the kubeadm tool to create and manage Kubernetes clusters. It performs the actions necessary to get a minimum viable, secure cluster up and running in a user friendly way.

Installing kubeadm shows you how to install kubeadm. Once installed, you can use it to create a cluster.

View kubeadm Install Guide

Install and Set Up kubectl on Linux

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.23 client can communicate with v1.22, v1.23, and v1.24 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on Linux

The following methods exist for installing kubectl on Linux:

- Install kubectl binary with curl on Linux
- Install using native package management
- Install using other package management

Install kubectl binary with curl on Linux

1. Download the latest release with the command:

   ```
   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
   dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
   ```

   Note:

   To download a specific version, replace the `$(curl -L -s https://dl.k8s.io/release/stable.txt)` portion of the command with the specific version.

   For example, to download version v1.23.0 on Linux, type:

   ```
   curl -LO https://dl.k8s.io/release/v1.23.0/bin/linux/amd64/
   kubectl
   ```

2. Validate the binary (optional)

   Download the kubectl checksum file:

   ```
   curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/
   release/stable.txt)/bin/linux/amd64/kubectl.sha256"
   ```

   Validate the kubectl binary against the checksum file:

   ```
   echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
   ```

   If valid, the output is:

   ```
   kubectl: OK
   ```

If the check fails, `sha256` exits with nonzero status and prints output similar to:

```
kubectl: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
```

Note: Download the same version of the binary and checksum.

3. Install kubectl

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/
kubectl
```

Note:

If you do not have root access on the target system, you can still install kubectl to the `~/.local/bin` directory:

```
chmod +x kubectl
mkdir -p ~/.local/bin
mv ./kubectl ~/.local/bin/kubectl
# and then append (or prepend) ~/.local/bin to $PATH
```

4. Test to ensure the version you installed is up-to-date:

```
kubectl version --client
```

Or use this for detailed view of version:

```
kubectl version --client --output=yaml
```

Install using native package management

- Debian-based distributions
- Red Hat-based distributions

1. Update the `apt` package index and install packages needed to use the Kubernetes `apt` repository:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates
curl
```

2. Download the Google Cloud public signing key:

```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-
keyring.gpg https://packages.cloud.google.com/apt/doc/apt-
key.gpg
```

3. Add the Kubernetes `apt` repository:

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-
keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial
main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

4. Update `apt` package index with the new repository and install kubectl:

   ```
   sudo apt-get update
   sudo apt-get install -y kubectl
   ```

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-
el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
sudo yum install -y kubectl
```

Install using other package management

- Snap
- Homebrew

If you are on Ubuntu or another Linux distribution that supports the snap package manager, kubectl is available as a snap application.

```
snap install kubectl --classic
kubectl version --client
```

If you are on Linux and using Homebrew package manager, kubectl is available for installation.

```
brew install kubectl
kubectl version --client
```

Verify kubectl configuration

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. By default, kubectl configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused -
did you specify the right host or port?
```

For example, if you are intending to run a Kubernetes cluster on your laptop (locally), you will need a tool like Minikube to be installed first and then re-run the commands stated above.

If kubectl cluster-info returns the url response but you can't access your cluster, to check whether it is configured properly, use:

```
kubectl cluster-info dump
```

Optional kubectl configurations and plugins

Enable shell autocompletion

kubectl provides autocompletion support for Bash, Zsh, Fish, and PowerShell, which can save you a lot of typing.

Below are the procedures to set up autocompletion for Bash, Fish, and Zsh.

- Bash
- Fish
- Zsh

Introduction

The kubectl completion script for Bash can be generated with the command `kubectl completion bash`. Sourcing the completion script in your shell enables kubectl autocompletion.

However, the completion script depends on **bash-completion**, which means that you have to install this software first (you can test if you have bash-completion already installed by running `type _init_completion`).

Install bash-completion

bash-completion is provided by many package managers (see here). You can install it with `apt-get install bash-completion` or `yum install bash-completion`, etc.

The above commands create `/usr/share/bash-completion/bash_completion`, which is the main script of bash-completion. Depending on your package manager, you have to manually source this file in your `~/.bashrc` file.

To find out, reload your shell and run `type _init_completion`. If the command succeeds, you're already set, otherwise add the following to your `~/.bashrc` file:

```
source /usr/share/bash-completion/bash_completion
```

Reload your shell and verify that bash-completion is correctly installed by typing `type _init_completion`.

Enable kubectl autocompletion

Bash

You now need to ensure that the kubectl completion script gets sourced in all your shell sessions. There are two ways in which you can do this:

- User
- System

```
echo 'source <(kubectl completion bash)' >>~/.bashrc
```

```
kubectl completion bash | sudo tee /etc/bash_completion.d/
kubectl > /dev/null
```

If you have an alias for kubectl, you can extend shell completion to work with that alias:

```
echo 'alias k=kubectl' >>~/.bashrc
echo 'complete -F __start_kubectl k' >>~/.bashrc
```

Note: bash-completion sources all completion scripts in `/etc/bash_completion.d`.

Both approaches are equivalent. After reloading your shell, kubectl autocompletion should be working.

The kubectl completion script for Fish can be generated with the command `kubectl completion fish`. Sourcing the completion script in your shell enables kubectl autocompletion.

To do so in all your shell sessions, add the following line to your `~/.config/fish/config.fish` file:

```
kubectl completion fish | source
```

After reloading your shell, kubectl autocompletion should be working.

The kubectl completion script for Zsh can be generated with the command `kubectl completion zsh`. Sourcing the completion script in your shell enables kubectl autocompletion.

To do so in all your shell sessions, add the following to your `~/.zshrc` file:

```
source <(kubectl completion zsh)
```

If you have an alias for kubectl, kubectl autocompletion will automatically work with it.

After reloading your shell, kubectl autocompletion should be working.

If you get an error like `2: command not found: compdef`, then add the following to the beginning of your `~/.zshrc` file:

```
autoload -Uz compinit
compinit
```

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit migrate to non deprecated apis

1. Download the latest release with the command:

   ```
   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert"
   ```

2. Validate the binary (optional)

   Download the kubectl-convert checksum file:

   ```
   curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl-convert.sha256"
   ```

   Validate the kubectl-convert binary against the checksum file:

   ```
   echo "$(cat kubectl-convert.sha256) kubectl-convert" | sha256sum --check
   ```

   If valid, the output is:

   ```
   kubectl-convert: OK
   ```

   If the check fails, `sha256` exits with nonzero status and prints output similar to:

   ```
   kubectl-convert: FAILED
   sha256sum: WARNING: 1 computed checksum did NOT match
   ```

   Note: Download the same version of the binary and checksum.

3. Install kubectl-convert

   ```
   sudo install -o root -g root -m 0755 kubectl-convert /usr/local/bin/kubectl-convert
   ```

4. Verify plugin is successfully installed

   ```
   kubectl convert --help
   ```

If you do not see an error, it means the plugin is successfully installed.

What's next

- Install Minikube
- See the getting started guides for more about creating clusters.
- Learn how to launch and expose your application.
- If you need access to a cluster you didn't create, see the Sharing Cluster Access document.
- Read the kubectl reference docs

Install and Set Up kubectl on macOS

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.23 client can communicate with v1.22, v1.23, and v1.24 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on macOS

The following methods exist for installing kubectl on macOS:

- Install kubectl binary with curl on macOS
- Install with Homebrew on macOS
- Install with Macports on macOS

Install kubectl binary with curl on macOS

1. Download the latest release:

   - Intel
   - Apple Silicon

   ```
   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl"
   ```

   ```
   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/arm64/kubectl"
   ```

   Note:

To download a specific version, replace the `$(curl -L -s https://dl.k8s.io/release/stable.txt)` portion of the command with the specific version.

For example, to download version v1.23.0 on Intel macOS, type:

```
curl -LO "https://dl.k8s.io/release/v1.23.0/bin/darwin/amd64/kubectl"
```

And for macOS on Apple Silicon, type:

```
curl -LO "https://dl.k8s.io/release/v1.23.0/bin/darwin/arm64/kubectl"
```

2. Validate the binary (optional)

   Download the kubectl checksum file:

   - Intel
   - Apple Silicon

```

   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl.sha256"

```

```

   curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/arm64/kubectl.sha256"

```

   Validate the kubectl binary against the checksum file:

```
echo "$(cat kubectl.sha256)  kubectl" | shasum -a 256 --check
```

   If valid, the output is:

```
kubectl: OK
```

   If the check fails, `shasum` exits with nonzero status and prints output similar to:

```
kubectl: FAILED
shasum: WARNING: 1 computed checksum did NOT match
```

   Note: Download the same version of the binary and checksum.

3. Make the kubectl binary executable.

```
chmod +x ./kubectl
```

4. Move the kubectl binary to a file location on your system `PATH`.

```
sudo mv ./kubectl /usr/local/bin/kubectl
sudo chown root: /usr/local/bin/kubectl
```

Note: Make sure `/usr/local/bin` is in your PATH environment variable.

5. Test to ensure the version you installed is up-to-date:

   ```
   kubectl version --client
   ```

   Or use this for detailed view of version:

   ```
   kubectl version --client --output=yaml
   ```

Install with Homebrew on macOS

If you are on macOS and using Homebrew package manager, you can install kubectl with Homebrew.

1. Run the installation command:

   ```
   brew install kubectl
   ```

   or

   ```
   brew install kubernetes-cli
   ```

2. Test to ensure the version you installed is up-to-date:

   ```
   kubectl version --client
   ```

Install with Macports on macOS

If you are on macOS and using Macports package manager, you can install kubectl with Macports.

1. Run the installation command:

   ```
   sudo port selfupdate
   sudo port install kubectl
   ```

2. Test to ensure the version you installed is up-to-date:

   ```
   kubectl version --client
   ```

Verify kubectl configuration

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. By default, kubectl configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused -
did you specify the right host or port?
```

For example, if you are intending to run a Kubernetes cluster on your laptop (locally), you will need a tool like Minikube to be installed first and then re-run the commands stated above.

If kubectl cluster-info returns the url response but you can't access your cluster, to check whether it is configured properly, use:

```
kubectl cluster-info dump
```

Optional kubectl configurations and plugins

Enable shell autocompletion

kubectl provides autocompletion support for Bash, Zsh, Fish, and PowerShell which can save you a lot of typing.

Below are the procedures to set up autocompletion for Bash, Fish, and Zsh.

- Bash
- Fish
- Zsh

Introduction

The kubectl completion script for Bash can be generated with `kubectl completion bash`. Sourcing this script in your shell enables kubectl completion.

However, the kubectl completion script depends on **bash-completion** which you thus have to previously install.

Warning: There are two versions of bash-completion, v1 and v2. V1 is for Bash 3.2 (which is the default on macOS), and v2 is for Bash 4.1+. The kubectl completion script **doesn't work** correctly with bash-completion v1 and Bash 3.2. It requires **bash-completion v2** and **Bash 4.1+**. Thus, to be able to correctly use kubectl completion on macOS, you have to install and use Bash 4.1+ (*instructions*). The following instructions assume that you use Bash 4.1+ (that is, any Bash version of 4.1 or newer).

Upgrade Bash

The instructions here assume you use Bash 4.1+. You can check your Bash's version by running:

```
echo $BASH_VERSION
```

If it is too old, you can install/upgrade it using Homebrew:

```
brew install bash
```

Reload your shell and verify that the desired version is being used:

```
echo $BASH_VERSION $SHELL
```

Homebrew usually installs it at `/usr/local/bin/bash`.

Install bash-completion

Note: As mentioned, these instructions assume you use Bash 4.1+, which means you will install bash-completion v2 (in contrast to Bash 3.2 and bash-completion v1, in which case kubectl completion won't work).

You can test if you have bash-completion v2 already installed with `type _init_completion`. If not, you can install it with Homebrew:

```
brew install bash-completion@2
```

As stated in the output of this command, add the following to your `~/.bash_profile` file:

```
export BASH_COMPLETION_COMPAT_DIR="/usr/local/etc/
bash_completion.d"
[[ -r "/usr/local/etc/profile.d/bash_completion.sh" ]] && . "/
usr/local/etc/profile.d/bash_completion.sh"
```

Reload your shell and verify that bash-completion v2 is correctly installed with `type _init_completion`.

Enable kubectl autocompletion

You now have to ensure that the kubectl completion script gets sourced in all your shell sessions. There are multiple ways to achieve this:

- Source the completion script in your `~/.bash_profile` file:

  ```
  echo 'source <(kubectl completion bash)' >>~/.bash_profile
  ```

- Add the completion script to the `/usr/local/etc/bash_completion.d` directory:

  ```
  kubectl completion bash >/usr/local/etc/bash_completion.d/
  kubectl
  ```

- If you have an alias for kubectl, you can extend shell completion to work with that alias:

  ```
  echo 'alias k=kubectl' >>~/.bash_profile
  echo 'complete -F __start_kubectl k' >>~/.bash_profile
  ```

- If you installed kubectl with Homebrew (as explained here), then the kubectl completion script should already be in `/usr/local/etc/bash_completion.d/kubectl`. In that case, you don't need to do anything.

  Note: The Homebrew installation of bash-completion v2 sources all the files in the `BASH_COMPLETION_COMPAT_DIR` directory, that's why the latter two methods work.

In any case, after reloading your shell, kubectl completion should be working.

The kubectl completion script for Fish can be generated with the command `kubectl completion fish`. Sourcing the completion script in your shell enables kubectl autocompletion.

To do so in all your shell sessions, add the following line to your `~/.config/fish/config.fish` file:

```
kubectl completion fish | source
```

After reloading your shell, kubectl autocompletion should be working.

The kubectl completion script for Zsh can be generated with the command `kubectl completion zsh`. Sourcing the completion script in your shell enables kubectl autocompletion.

To do so in all your shell sessions, add the following to your `~/.zshrc` file:

```
source <(kubectl completion zsh)
```

If you have an alias for kubectl, kubectl autocompletion will automatically work with it.

After reloading your shell, kubectl autocompletion should be working.

If you get an error like `2: command not found: compdef`, then add the following to the beginning of your `~/.zshrc` file:

```
autoload -Uz compinit
compinit
```

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit [migrate to non deprecated apis]

1. Download the latest release with the command:

   - Intel
   - Apple Silicon

   ```
      curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
   dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl-
   convert"
   ```

   ```
      curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
   dl.k8s.io/release/stable.txt)/bin/darwin/arm64/kubectl-
   convert"
   ```

2. Validate the binary (optional)

   Download the kubectl-convert checksum file:

   - Intel
   - Apple Silicon

   ```
      curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
   dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl-
   convert.sha256"
   ```

   ```
      curl -LO "https://dl.k8s.io/release/$(curl -L -s https://
   dl.k8s.io/release/stable.txt)/bin/darwin/arm64/kubectl-
   convert.sha256"
   ```

   Validate the kubectl-convert binary against the checksum file:

   ```
   echo "$(cat kubectl-convert.sha256)  kubectl-convert" |
   shasum -a 256 --check
   ```

   If valid, the output is:

   ```
   kubectl-convert: OK
   ```

   If the check fails, `shasum` exits with nonzero status and prints output similar to:

   ```
   kubectl-convert: FAILED
   shasum: WARNING: 1 computed checksum did NOT match
   ```

Note: Download the same version of the binary and checksum.

3. Make kubectl-convert binary executable

   ```
   chmod +x ./kubectl-convert
   ```

4. Move the kubectl-convert binary to a file location on your system `PATH`.

   ```
   sudo mv ./kubectl-convert /usr/local/bin/kubectl-convert
   sudo chown root: /usr/local/bin/kubectl-convert
   ```

   Note: Make sure `/usr/local/bin` is in your PATH environment variable.

5. Verify plugin is successfully installed

   ```
   kubectl convert --help
   ```

   If you do not see an error, it means the plugin is successfully installed.

What's next

- Install Minikube
- See the getting started guides for more about creating clusters.
- Learn how to launch and expose your application.
- If you need access to a cluster you didn't create, see the Sharing Cluster Access document.
- Read the kubectl reference docs

Install and Set Up kubectl on Windows

Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.23 client can communicate with v1.22, v1.23, and v1.24 control planes. Using the latest compatible version of kubectl helps avoid unforeseen issues.

Install kubectl on Windows

The following methods exist for installing kubectl on Windows:

- Install kubectl binary with curl on Windows
- Install on Windows using Chocolatey or Scoop

Install kubectl binary with curl on Windows

1. Download the latest release v1.23.0.

Or if you have `curl` installed, use this command:

```
curl -LO "https://dl.k8s.io/release/v1.23.0/bin/windows/
amd64/kubectl.exe"
```

Note: To find out the latest stable version (for example, for scripting), take a look at https://dl.k8s.io/release/stable.txt.

2. Validate the binary (optional)

   Download the `kubectl` checksum file:

   ```
   curl -LO "https://dl.k8s.io/v1.23.0/bin/windows/amd64/
   kubectl.exe.sha256"
   ```

   Validate the `kubectl` binary against the checksum file:

   - Using Command Prompt to manually compare `CertUtil`'s output to the checksum file downloaded:

     ```
     CertUtil -hashfile kubectl.exe SHA256
     type kubectl.exe.sha256
     ```

   - Using PowerShell to automate the verification using the `-eq` operator to get a `True` or `False` result:

     ```
     $($(CertUtil -hashfile .\kubectl.exe SHA256)[1] -replace
     " ", "") -eq $(type .\kubectl.exe.sha256)
     ```

3. Append or prepend the `kubectl` binary folder to your `PATH` environment variable.

4. Test to ensure the version of `kubectl` is the same as downloaded:

   ```
   kubectl version --client
   ```

   Or use this for detailed view of version:

   ```
   kubectl version --client --output=yaml
   ```

Note: Docker Desktop for Windows adds its own version of `kubectl` to `PATH`. If you have installed Docker Desktop before, you may need to place your `PATH` entry before the one added by the Docker Desktop installer or remove the Docker Desktop's `kubectl`.

Install on Windows using Chocolatey or Scoop

1. To install kubectl on Windows you can use either Chocolatey package manager or Scoop command-line installer.

   - choco
   - scoop

   ```
   choco install kubernetes-cli
   ```

```
scoop install kubectl
```

2. Test to ensure the version you installed is up-to-date:

   ```
   kubectl version --client
   ```

3. Navigate to your home directory:

   ```
   # If you're using cmd.exe, run: cd %USERPROFILE%
   cd ~
   ```

4. Create the `.kube` directory:

   ```
   mkdir .kube
   ```

5. Change to the `.kube` directory you just created:

   ```
   cd .kube
   ```

6. Configure kubectl to use a remote Kubernetes cluster:

   ```
   New-Item config -type file
   ```

Note: Edit the config file with a text editor of your choice, such as Notepad.

Verify kubectl configuration

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. By default, kubectl configuration is located at `~/.kube/config`.

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 configured correctly or is not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused -
did you specify the right host or port?
```

For example, if you are intending to run a Kubernetes cluster on your laptop (locally), you will need a tool like Minikube to be installed first and then re-run the commands stated above.

If kubectl cluster-info returns the url response but you can't access your cluster, to check whether it is configured properly, use:

```
kubectl cluster-info dump
```

Optional kubectl configurations and plugins

Enable shell autocompletion

kubectl provides autocompletion support for Bash, Zsh, Fish, and PowerShell, which can save you a lot of typing.

Below are the procedures to set up autocompletion for PowerShell.

The kubectl completion script for PowerShell can be generated with the command `kubectl completion powershell`.

To do so in all your shell sessions, add the following line to your `$PROFILE` file:

```
kubectl completion powershell | Out-String | Invoke-Expression
```

This command will regenerate the auto-completion script on every PowerShell start up. You can also add the generated script directly to your `$PROFILE` file.

To add the generated script to your `$PROFILE` file, run the following line in your powershell prompt:

```
kubectl completion powershell >> $PROFILE
```

After reloading your shell, kubectl autocompletion should be working.

Install `kubectl convert` plugin

A plugin for Kubernetes command-line tool `kubectl`, which allows you to convert manifests between different API versions. This can be particularly helpful to migrate manifests to a non-deprecated api version with newer Kubernetes release. For more info, visit migrate to non deprecated apis

1. Download the latest release with the command:

   ```
   curl -LO "https://dl.k8s.io/release/v1.23.0/bin/windows/amd64/kubectl-convert.exe"
   ```

2. Validate the binary (optional)

   Download the `kubectl-convert` checksum file:

   ```
   curl -LO "https://dl.k8s.io/v1.23.0/bin/windows/amd64/kubectl-convert.exe.sha256"
   ```

   Validate the `kubectl-convert` binary against the checksum file:

   - Using Command Prompt to manually compare `CertUtil`'s output to the checksum file downloaded:

```
CertUtil -hashfile kubectl-convert.exe SHA256
type kubectl-convert.exe.sha256
```

    - Using PowerShell to automate the verification using the `-eq` operator to get a `True` or `False` result:

```
$($(CertUtil -hashfile .\kubectl-convert.exe SHA256)[1]
-replace " ", "") -eq $(type .\kubectl-convert.exe.sha25
6)
```

3. Append or prepend the `kubectl-convert` binary folder to your `PATH` environment variable.

4. Verify plugin is successfully installed

```
kubectl convert --help
```

   If you do not see an error, it means the plugin is successfully installed.

What's next

- Install Minikube
- See the getting started guides for more about creating clusters.
- Learn how to launch and expose your application.
- If you need access to a cluster you didn't create, see the Sharing Cluster Access document.
- Read the kubectl reference docs

Administer a Cluster

Learn common tasks for administering a cluster.

Administration with kubeadm

Migrating from dockershim

Certificates

Manage Memory, CPU, and API Resources

Install a Network Policy Provider

Access Clusters Using the Kubernetes API

Advertise Extended Resources for a Node

Autoscale the DNS Service in a Cluster

Change the default StorageClass

Change the Reclaim Policy of a PersistentVolume

Cloud Controller Manager Administration

Configure Quotas for API Objects

Control CPU Management Policies on the Node

Control Topology Management Policies on a node

Customizing DNS Service

Debugging DNS Resolution

Declare Network Policy

Developing Cloud Controller Manager

Enable Or Disable A Kubernetes API

Enabling Service Topology

Encrypting Secret Data at Rest

Guaranteed Scheduling For Critical Add-On Pods

IP Masquerade Agent User Guide

Limit Storage Consumption

Migrate Replicated Control Plane To Use Cloud Controller Manager

Namespaces Walkthrough

Operating etcd clusters for Kubernetes

Reconfigure a Node's Kubelet in a Live Cluster

Reserve Compute Resources for System Daemons

Running Kubernetes Node Components as a Non-root User

Safely Drain a Node

Securing a Cluster

Set Kubelet parameters via a config file

Share a Cluster with Namespaces

Upgrade A Cluster

Use Cascading Deletion in a Cluster

Using a KMS provider for data encryption

Using CoreDNS for Service Discovery

Using NodeLocal DNSCache in Kubernetes clusters

Using sysctls in a Kubernetes Cluster

Utilizing the NUMA-aware Memory Manager

Administration with kubeadm

Certificate Management with kubeadm

Configuring a cgroup driver

Reconfiguring a kubeadm cluster

Upgrading kubeadm clusters

Adding Windows nodes

Upgrading Windows nodes

Certificate Management with kubeadm

FEATURE STATE: `Kubernetes v1.15 [stable]`

Client certificates generated by kubeadm expire after 1 year. This page explains how to manage certificate renewals with kubeadm. It also covers other tasks related to kubeadm certificate management.

Before you begin

You should be familiar with PKI certificates and requirements in Kubernetes.

Using custom certificates

By default, kubeadm generates all the certificates needed for a cluster to run. You can override this behavior by providing your own certificates.

To do so, you must place them in whatever directory is specified by the `--cert-dir` flag or the `certificatesDir` field of kubeadm's `ClusterConfiguration`. By default this is `/etc/kubernetes/pki`.

If a given certificate and private key pair exists before running `kubeadm init`, kubeadm does not overwrite them. This means you can, for example, copy an existing CA into `/etc/kubernetes/pki/ca.crt` and `/etc/kubernetes/pki/ca.key`, and kubeadm will use this CA for signing the rest of the certificates.

External CA mode

It is also possible to provide only the `ca.crt` file and not the `ca.key` file (this is only available for the root CA file, not other cert pairs). If all other certificates and kubeconfig files are in place, kubeadm recognizes this condition and activates the "External CA" mode. kubeadm will proceed without the CA key on disk.

Instead, run the controller-manager standalone with `--controllers=csrsigner` and point to the CA certificate and key.

PKI certificates and requirements includes guidance on setting up a cluster to use an external CA.

Check certificate expiration

You can use the `check-expiration` subcommand to check when certificates expire:

```
kubeadm certs check-expiration
```

The output is similar to this:

```
CERTIFICATE                EXPIRES                  RESIDUAL
TIME   CERTIFICATE AUTHORITY   EXTERNALLY MANAGED
admin.conf                 Dec 30, 2020 23:36 UTC
364d                            no
apiserver                  Dec 30, 2020 23:36 UTC
364d            ca              no
apiserver-etcd-client      Dec 30, 2020 23:36 UTC
364d            etcd-ca         no
apiserver-kubelet-client   Dec 30, 2020 23:36 UTC
364d            ca              no
controller-manager.conf    Dec 30, 2020 23:36 UTC
364d                            no
etcd-healthcheck-client    Dec 30, 2020 23:36 UTC
364d            etcd-ca         no
etcd-peer                  Dec 30, 2020 23:36 UTC
364d            etcd-ca         no
etcd-server                Dec 30, 2020 23:36 UTC
364d            etcd-ca         no
```

```
front-proxy-client         Dec 30, 2020 23:36 UTC
364d            front-proxy-ca          no
scheduler.conf             Dec 30, 2020 23:36 UTC
364d                                    no

CERTIFICATE AUTHORITY   EXPIRES                  RESIDUAL TIME
EXTERNALLY MANAGED
ca                      Dec 28, 2029 23:36 UTC   9y
no
etcd-ca                 Dec 28, 2029 23:36 UTC   9y
no
front-proxy-ca          Dec 28, 2029 23:36 UTC   9y
no
```

The command shows expiration/residual time for the client certificates in the `/etc/kubernetes/pki` folder and for the client certificate embedded in the KUBECONFIG files used by kubeadm (`admin.conf`, `controller-manager.conf` and `scheduler.conf`).

Additionally, kubeadm informs the user if the certificate is externally managed; in this case, the user should take care of managing certificate renewal manually/using other tools.

Warning: `kubeadm` cannot manage certificates signed by an external CA.
Note: `kubelet.conf` is not included in the list above because kubeadm configures kubelet for automatic certificate renewal with rotatable certificates under `/var/lib/kubelet/pki`. To repair an expired kubelet client certificate see Kubelet client certificate rotation fails.
Warning:

On nodes created with `kubeadm init`, prior to kubeadm version 1.17, there is a bug where you manually have to modify the contents of `kubelet.conf`. After `kubeadm init` finishes, you should update `kubelet.conf` to point to the rotated kubelet client certificates, by replacing `client-certificate-data` and `client-key-data` with:

```
client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
```

Automatic certificate renewal

kubeadm renews all the certificates during control plane upgrade.

This feature is designed for addressing the simplest use cases; if you don't have specific requirements on certificate renewal and perform Kubernetes version upgrades regularly (less than 1 year in between each upgrade), kubeadm will take care of keeping your cluster up to date and reasonably secure.

Note: It is a best practice to upgrade your cluster frequently in order to stay secure.

If you have more complex requirements for certificate renewal, you can opt out from the default behavior by passing `--certificate-renewal=false` to `kubeadm upgrade apply` or to `kubeadm upgrade node`.

Warning: Prior to kubeadm version 1.17 there is a bug where the default value for `--certificate-renewal` is `false` for the `kubeadm upgrade node` command. In that case, you should explicitly set `--certificate-renewal=true`.

Manual certificate renewal

You can renew your certificates manually at any time with the `kubeadm certs renew` command.

This command performs the renewal using CA (or front-proxy-CA) certificate and key stored in `/etc/kubernetes/pki`.

After running the command you should restart the control plane Pods. This is required since dynamic certificate reload is currently not supported for all components and certificates. Static Pods are managed by the local kubelet and not by the API Server, thus kubectl cannot be used to delete and restart them. To restart a static Pod you can temporarily remove its manifest file from `/etc/kubernetes/manifests/` and wait for 20 seconds (see the `fileCheckFrequency` value in KubeletConfiguration struct. The kubelet will terminate the Pod if it's no longer in the manifest directory. You can then move the file back and after another `fileCheckFrequency` period, the kubelet will recreate the Pod and the certificate renewal for the component can complete.

Warning: If you are running an HA cluster, this command needs to be executed on all the control-plane nodes.
Note: `certs renew` uses the existing certificates as the authoritative source for attributes (Common Name, Organization, SAN, etc.) instead of the kubeadm-config ConfigMap. It is strongly recommended to keep them both in sync.

`kubeadm certs renew` provides the following options:

The Kubernetes certificates normally reach their expiration date after one year.

- `--csr-only` can be used to renew certificates with an external CA by generating certificate signing requests (without actually renewing certificates in place); see next paragraph for more information.
- It's also possible to renew a single certificate instead of all.

Renew certificates with the Kubernetes certificates API

This section provides more details about how to execute manual certificate renewal using the Kubernetes certificates API.

Caution: These are advanced topics for users who need to integrate their organization's certificate infrastructure into a kubeadm-built cluster. If the default kubeadm configuration satisfies your needs, you should let kubeadm manage certificates instead.

Set up a signer

The Kubernetes Certificate Authority does not work out of the box. You can configure an external signer such as cert-manager, or you can use the built-in signer.

The built-in signer is part of `kube-controller-manager`.

To activate the built-in signer, you must pass the `--cluster-signing-cert-file` and `--cluster-signing-key-file` flags.

If you're creating a new cluster, you can use a kubeadm configuration file:

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    cluster-signing-cert-file: /etc/kubernetes/pki/ca.crt
    cluster-signing-key-file: /etc/kubernetes/pki/ca.key
```

Create certificate signing requests (CSR)

See Create CertificateSigningRequest for creating CSRs with the Kubernetes API.

Renew certificates with external CA

This section provide more details about how to execute manual certificate renewal using an external CA.

To better integrate with external CAs, kubeadm can also produce certificate signing requests (CSRs). A CSR represents a request to a CA for a signed certificate for a client. In kubeadm terms, any certificate that would normally be signed by an on-disk CA can be produced as a CSR instead. A CA, however, cannot be produced as a CSR.

Create certificate signing requests (CSR)

You can create certificate signing requests with `kubeadm certs renew --csr-only`.

Both the CSR and the accompanying private key are given in the output. You can pass in a directory with `--csr-dir` to output the CSRs to the specified location. If `--csr-dir` is not specified, the default certificate directory (`/etc/kubernetes/pki`) is used.

Certificates can be renewed with `kubeadm certs renew --csr-only`. As with `kubeadm init`, an output directory can be specified with the `--csr-dir` flag.

A CSR contains a certificate's name, domains, and IPs, but it does not specify usages. It is the responsibility of the CA to specify the correct cert usages when issuing a certificate.

- In `openssl` this is done with the `openssl ca` command.
- In `cfssl` you specify usages in the config file.

After a certificate is signed using your preferred method, the certificate and the private key must be copied to the PKI directory (by default `/etc/kubernetes/pki`).

Certificate authority (CA) rotation

Kubeadm does not support rotation or replacement of CA certificates out of the box.

For more information about manual rotation or replacement of CA, see manual rotation of CA certificates.

Enabling signed kubelet serving certificates

By default the kubelet serving certificate deployed by kubeadm is self-signed. This means a connection from external services like the metrics-server to a kubelet cannot be secured with TLS.

To configure the kubelets in a new kubeadm cluster to obtain properly signed serving certificates you must pass the following minimal configuration to `kubeadm init`:

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
serverTLSBootstrap: true
```

If you have already created the cluster you must adapt it by doing the following:

- Find and edit the `kubelet-config-1.23` ConfigMap in the `kube-system` namespace. In that ConfigMap, the `kubelet` key has a KubeletConfiguration document as its value. Edit the KubeletConfiguration document to set `serverTLSBootstrap: true`.
- On each node, add the `serverTLSBootstrap: true` field in `/var/lib/kubelet/config.yaml` and restart the kubelet with `systemctl restart kubelet`

The field `serverTLSBootstrap: true` will enable the bootstrap of kubelet serving certificates by requesting them from the `certificates.k8s.io` API. One known limitation is that the CSRs (Certificate Signing Requests) for these certificates cannot be automatically approved by the default signer in the kube-controller-manager - `kubernetes.io/kubelet-serving`. This will require action from the user or a third party controller.

These CSRs can be viewed using:

```
kubectl get csr
NAME        AGE     SIGNERNAME
REQUESTOR                      CONDITION
csr-9wvgt   112s    kubernetes.io/kubelet-serving
system:node:worker-1           Pending
csr-lz97v   1m58s   kubernetes.io/kubelet-serving
system:node:control-plane-1    Pending
```

To approve them you can do the following:

```
kubectl certificate approve <CSR-name>
```

By default, these serving certificate will expire after one year. Kubeadm sets the `KubeletConfiguration` field `rotateCertificates` to `true`, which means that close to expiration a new set of CSRs for the serving certificates will be created and must be approved to complete the rotation. To understand more see Certificate Rotation.

If you are looking for a solution for automatic approval of these CSRs it is recommended that you contact your cloud provider and ask if they have a CSR signer that verifies the node identity with an out of band mechanism.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Third party custom controllers can be used:

- kubelet-csr-approver

Such a controller is not a secure mechanism unless it not only verifies the CommonName in the CSR but also verifies the requested IPs and domain names. This would prevent a malicious actor that has access to a kubelet client certificate to create CSRs requesting serving certificates for any IP or domain name.

Generating kubeconfig files for additional users

During cluster creation, kubeadm signs the certificate in the `admin.conf` to have `Subject: O = system:masters, CN = kubernetes-admin`. `system:masters` is a break-glass, super user group that bypasses the authorization layer (e.g. RBAC). Sharing the `admin.conf` with additional users is **not recommended**!

Instead, you can use the `kubeadm kubeconfig user` command to generate kubeconfig files for additional users. The command accepts a mixture of command line flags and kubeadm configuration options. The generated kubeconfig will be written to stdout and can be piped to a file using `kubeadm kubeconfig user ... > somefile.conf`.

Example configuration file that can be used with `--config`:

```yaml
# example.yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
# Will be used as the target "cluster" in the kubeconfig
clusterName: "kubernetes"
# Will be used as the "server" (IP or DNS name) of this cluster
in the kubeconfig
controlPlaneEndpoint: "some-dns-address:6443"
# The cluster CA key and certificate will be loaded from this
local directory
certificatesDir: "/etc/kubernetes/pki"
```

Make sure that these settings match the desired target cluster settings. To see the settings of an existing cluster use:

```
kubectl get cm kubeadm-config -n kube-system -o=jsonpath="{.data.
ClusterConfiguration}"
```

The following example will generate a kubeconfig file with credentials valid for 24 hours for a new user `johndoe` that is part of the `appdevs` group:

```
kubeadm kubeconfig user --config example.yaml --org appdevs --
client-name johndoe --validity-period 24h
```

The following example will generate a kubeconfig file with administrator credentials valid for 1 week:

```
kubeadm kubeconfig user --config example.yaml --client-name
admin --validity-period 168h
```

Configuring a cgroup driver

This page explains how to configure the kubelet cgroup driver to match the container runtime cgroup driver for kubeadm clusters.

Before you begin

You should be familiar with the Kubernetes container runtime requirements.

Configuring the container runtime cgroup driver

The Container runtimes page explains that the `systemd` driver is recommended for kubeadm based setups instead of the `cgroupfs` driver, because kubeadm manages the kubelet as a systemd service.

The page also provides details on how to setup a number of different container runtimes with the `systemd` driver by default.

Configuring the kubelet cgroup driver

kubeadm allows you to pass a `KubeletConfiguration` structure during `kubeadm init`. This `KubeletConfiguration` can include the `cgroupDriver` field which controls the cgroup driver of the kubelet.

Note: In v1.22, if the user is not setting the `cgroupDriver` field under `KubeletConfiguration`, `kubeadm` will default it to `systemd`.

A minimal example of configuring the field explicitly:

```yaml
# kubeadm-config.yaml
kind: ClusterConfiguration
apiVersion: kubeadm.k8s.io/v1beta3
kubernetesVersion: v1.21.0
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
```

Such a configuration file can then be passed to the kubeadm command:

```
kubeadm init --config kubeadm-config.yaml
```

Note:

Kubeadm uses the same `KubeletConfiguration` for all nodes in the cluster. The `KubeletConfiguration` is stored in a ConfigMap object under the `kube-system` namespace.

Executing the sub commands `init`, `join` and `upgrade` would result in kubeadm writing the `KubeletConfiguration` as a file under `/var/lib/kubelet/config.yaml` and passing it to the local node kubelet.

Using the `cgroupfs` driver

As this guide explains using the `cgroupfs` driver with kubeadm is not recommended.

To continue using `cgroupfs` and to prevent `kubeadm upgrade` from modifying the `KubeletConfiguration` cgroup driver on existing setups, you must be explicit about its value. This applies to a case where you do not wish future versions of kubeadm to apply the `systemd` driver by default.

See the below section on "Modify the kubelet ConfigMap" for details on how to be explicit about the value.

If you wish to configure a container runtime to use the `cgroupfs` driver, you must refer to the documentation of the container runtime of your choice.

Migrating to the `systemd` driver

To change the cgroup driver of an existing kubeadm cluster to `systemd` in-place, a similar procedure to a kubelet upgrade is required. This must include both steps outlined below.

Note: Alternatively, it is possible to replace the old nodes in the cluster with new ones that use the `systemd` driver. This requires executing only the first step below before joining the new nodes and ensuring the workloads can safely move to the new nodes before deleting the old nodes.

Modify the kubelet ConfigMap

- Find the kubelet ConfigMap name using `kubectl get cm -n kube-system | grep kubelet-config`.
- Call `kubectl edit cm kubelet-config-x.yy -n kube-system` (replace `x.yy` with the Kubernetes version).
- Either modify the existing `cgroupDriver` value or add a new field that looks like this:

  ```yaml
  cgroupDriver: systemd
  ```

  This field must be present under the `kubelet:` section of the ConfigMap.

Update the cgroup driver on all nodes

For each node in the cluster:

- Drain the node using `kubectl drain <node-name> --ignore-daemonsets`
- Stop the kubelet using `systemctl stop kubelet`
- Stop the container runtime
- Modify the container runtime cgroup driver to `systemd`
- Set `cgroupDriver: systemd` in `/var/lib/kubelet/config.yaml`
- Start the container runtime
- Start the kubelet using `systemctl start kubelet`
- Uncordon the node using `kubectl uncordon <node-name>`

Execute these steps on nodes one at a time to ensure workloads have sufficient time to schedule on different nodes.

Once the process is complete ensure that all nodes and workloads are healthy.

Reconfiguring a kubeadm cluster

kubeadm does not support automated ways of reconfiguring components that were deployed on managed nodes. One way of automating this would be by using a custom operator.

To modify the components configuration you must manually edit associated cluster objects and files on disk.

This guide shows the correct sequence of steps that need to be performed to achieve kubeadm cluster reconfiguration.

Before you begin

- You need a cluster that was deployed using kubeadm
- Have administrator credentials (`/etc/kubernetes/admin.conf`) and network connectivity to a running kube-apiserver in the cluster from a host that has kubectl installed
- Have a text editor installed on all hosts

Reconfiguring the cluster

kubeadm writes a set of cluster wide component configuration options in ConfigMaps and other objects. These objects must be manually edited. The command `kubectl edit` can be used for that.

The `kubectl edit` command will open a text editor where you can edit and save the object directly.

You can use the environment variables `KUBECONFIG` and `KUBE_EDITOR` to specify the location of the kubectl consumed kubeconfig file and preferred text editor.

For example:

```
KUBECONFIG=/etc/kubernetes/admin.conf KUBE_EDITOR=nano kubectl
edit <parameters>
```

Note: Upon saving any changes to these cluster objects, components running on nodes may not be automatically updated. The steps below instruct you on how to perform that manually.
Warning: Component configuration in ConfigMaps is stored as unstructured data (YAML string). This means that validation will not be performed upon updating the contents of a ConfigMap. You have to be careful to follow the documented API format for a particular component configuration and avoid introducing typos and YAML indentation mistakes.

Applying cluster configuration changes

Updating the `ClusterConfiguration`

During cluster creation and upgrade, kubeadm writes its `ClusterConfiguration` in a ConfigMap called `kubeadm-config` in the `kube-system` namespace.

To change a particular option in the `ClusterConfiguration` you can edit the ConfigMap with this command:

```
kubectl edit cm -n kube-system kubeadm-config
```

The configuration is located under the `data.ClusterConfiguration` key.

Note: The `ClusterConfiguration` includes a variety of options that affect the configuration of individual components such as kube-apiserver, kube-scheduler, kube-controller-manager, CoreDNS, etcd and kube-proxy. Changes to the configuration must be reflected on node components manually.

Reflecting `ClusterConfiguration` changes on control plane nodes

kubeadm manages the control plane components as static Pod manifests located in the directory `/etc/kubernetes/manifests`. Any changes to the `ClusterConfiguration` under the `apiServer`, `controllerManager`, `scheduler` or `etcd` keys must be reflected in the associated files in the manifests directory on a control plane node.

Such changes may include:

- `extraArgs` - requires updating the list of flags passed to a component container
- `extraMounts` - requires updated the volume mounts for a component container

- `*SANs` - requires writing new certificates with updated Subject Alternative Names.

Before proceeding with these changes, make sure you have backed up the directory `/etc/kubernetes/`.

To write new certificates you can use:

```
kubeadm init phase certs <component-name> --config <config-file>
```

To write new manifest files in `/etc/kubernetes/manifests` you can use:

```
kubeadm init phase control-plane <component-name> --config
<config-file>
```

The `<config-file>` contents must match the updated `ClusterConfiguration`. The `<component-name>` value must be the name of the component.

Note: Updating a file in `/etc/kubernetes/manifests` will tell the kubelet to restart the static Pod for the corresponding component. Try doing these changes one node at a time to leave the cluster without downtime.

Applying kubelet configuration changes

Updating the `KubeletConfiguration`

During cluster creation and upgrade, kubeadm writes its `KubeletConfiguration` in a ConfigMap called `kubelet-config` in the `kube-system` namespace.

You can edit the ConfigMap with this command:

```
kubectl edit cm -n kube-system kubelet-config
```

The configuration is located under the `data.kubelet` key.

Reflecting the kubelet changes

To reflect the change on kubeadm nodes you must do the following:

- Log in to a kubeadm node
- Run `kubeadm upgrade node phase kubelet-config` to download the latest `kubelet-config` ConfigMap contents into the local file `/var/lib/kubelet/config.conf`
- Edit the file `/var/lib/kubelet/kubeadm-flags.env` to apply additional configuration with flags
- Restart the kubelet service with `systemctl restart kubelet`

Note: Do these changes one node at a time to allow workloads to be rescheduled properly.
Note: During `kubeadm upgrade`, kubeadm downloads the `KubeletConfiguration` from the `kubelet-config` ConfigMap and overwrite the contents of `/var/lib/kubelet/config.conf`. This means that node local configuration must be applied either by flags in `/var/lib/kubelet/kubeadm-flags.env` or

by manually updating the contents of `/var/lib/kubelet/config.conf` after `kubeadm upgrade`, and then restarting the kubelet.

Applying kube-proxy configuration changes

Updating the `KubeProxyConfiguration`

During cluster creation and upgrade, kubeadm writes its `KubeProxyConfiguration` in a ConfigMap in the `kube-system` namespace called `kube-proxy`.

This ConfigMap is used by the `kube-proxy` DaemonSet in the `kube-system` namespace.

To change a particular option in the `KubeProxyConfiguration`, you can edit the ConfigMap with this command:

```
kubectl edit cm -n kube-system kube-proxy
```

The configuration is located under the `data.config.conf` key.

Reflecting the kube-proxy changes

Once the `kube-proxy` ConfigMap is updated, you can restart all kube-proxy Pods:

Obtain the Pod names:

```
kubectl get po -n kube-system | grep kube-proxy
```

Delete a Pod with:

```
kubectl delete po -n kube-system <pod-name>
```

New Pods that use the updated ConfigMap will be created.

Note: Because kubeadm deploys kube-proxy as a DaemonSet, node specific configuration is unsupported.

Applying CoreDNS configuration changes

Updating the CoreDNS Deployment and Service

kubeadm deploys CoreDNS as a Deployment called `coredns` and with a Service `kube-dns`, both in the `kube-system` namespace.

To update any of the CoreDNS settings, you can edit the Deployment and Service objects:

```
kubectl edit deployment -n kube-system coredns
kubectl edit service -n kube-system kube-dns
```

Reflecting the CoreDNS changes

Once the CoreDNS changes are applied you can delete the CoreDNS Pods:

Obtain the Pod names:

```
kubectl get po -n kube-system | grep coredns
```

Delete a Pod with:

```
kubectl delete po -n kube-system <pod-name>
```

New Pods with the updated CoreDNS configuration will be created.

Note: kubeadm does not allow CoreDNS configuration during cluster creation and upgrade. This means that if you execute `kubeadm upgrade apply`, your changes to the CoreDNS objects will be lost and must be reapplied.

Persisting the reconfiguration

During the execution of `kubeadm upgrade` on a managed node, kubeadm might overwrite configuration that was applied after the cluster was created (reconfiguration).

Persisting Node object reconfiguration

kubeadm writes Labels, Taints, CRI socket and other information on the Node object for a particular Kubernetes node. To change any of the contents of this Node object you can use:

```
kubectl edit no <node-name>
```

During `kubeadm upgrade` the contents of such a Node might get overwritten. If you would like to persist your modifications to the Node object after upgrade, you can prepare a kubectl patch and apply it to the Node object:

```
kubectl patch no <node-name> --patch-file <patch-file>
```

Persisting control plane component reconfiguration

The main source of control plane configuration is the `ClusterConfiguration` object stored in the cluster. To extend the static Pod manifests configuration, patches can be used.

These patch files must remain as files on the control plane nodes to ensure that they can be used by the `kubeadm upgrade ... --patches <directory>`.

If reconfiguration is done to the `ClusterConfiguration` and static Pod manifests on disk, the set of node specific patches must be updated accordingly.

Persisting kubelet reconfiguration

Any changes to the `KubeletConfiguration` stored in `/var/lib/kubelet/config.conf` will be overwritten on `kubeadm upgrade` by downloading the contents of the cluster wide `kubelet-config` ConfigMap. To persist kubelet node specific configuration either the file `/var/lib/kubelet/config.conf` has to be updated manually post-upgrade or the file `/var/lib/kubelet/kubeadm-flags.env` can include flags. The kubelet flags override the associated `KubeletConfiguration` options, but note that some of the flags are deprecated.

A kubelet restart will be required after changing `/var/lib/kubelet/config.conf` or `/var/lib/kubelet/kubeadm-flags.env`.

What's next

- Upgrading kubeadm clusters
- Customizing components with the kubeadm API
- Certificate management with kubeadm

Upgrading kubeadm clusters

This page explains how to upgrade a Kubernetes cluster created with kubeadm from version 1.22.x to version 1.23.x, and from version 1.23.x to 1.23.y (where `y > x`). Skipping MINOR versions when upgrading is unsupported.

To see information about upgrading clusters created using older versions of kubeadm, please refer to following pages instead:

- Upgrading a kubeadm cluster from 1.21 to 1.22
- Upgrading a kubeadm cluster from 1.20 to 1.21
- Upgrading a kubeadm cluster from 1.19 to 1.20
- Upgrading a kubeadm cluster from 1.18 to 1.19

The upgrade workflow at high level is the following:

1. Upgrade a primary control plane node.
2. Upgrade additional control plane nodes.
3. Upgrade worker nodes.

Before you begin

- Make sure you read the release notes carefully.
- The cluster should use a static control plane and etcd pods or external etcd.
- Make sure to back up any important components, such as app-level state stored in a database. `kubeadm upgrade` does not touch your workloads, only components internal to Kubernetes, but backups are always a best practice.
- Swap must be disabled.

Additional information

- The instructions below outline when to drain each node during the upgrade process. If you are performing a **minor** version upgrade for any kubelet, you **must** first drain the node (or nodes) that you are upgrading. In the case of control plane nodes, they could be running CoreDNS Pods or other critical workloads. For more information see Draining nodes.
- All containers are restarted after upgrade, because the container spec hash value is changed.
- To verify that the kubelet service has successfully restarted after the kubelet has been upgraded, you can execute `systemctl status kubelet` or view the service logs with `journalctl -xeu kubelet`.
- Usage of the `--config` flag of `kubeadm upgrade` with kubeadm configuration API types with the purpose of reconfiguring the cluster is not recommended and can have unexpected results. Follow the steps in Reconfiguring a kubeadm cluster instead.

Determine which version to upgrade to

Find the latest patch release for Kubernetes 1.23 using the OS package manager:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora

```
apt update
apt-cache madison kubeadm
# find the latest 1.23 version in the list
# it should look like 1.23.x-00, where x is the latest patch
```

```
yum list --showduplicates kubeadm --disableexcludes=kubernetes
# find the latest 1.23 version in the list
# it should look like 1.23.x-0, where x is the latest patch
```

Upgrading control plane nodes

The upgrade procedure on control plane nodes should be executed one node at a time. Pick a control plane node that you wish to upgrade first. It must have the `/etc/kubernetes/admin.conf` file.

Call "kubeadm upgrade"

For the first control plane node

- Upgrade kubeadm:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.23.x-00 && \
apt-mark hold kubeadm
```

```
# replace x in 1.23.x-0 with the latest patch version
yum install -y kubeadm-1.23.x-0 --disableexcludes=kubernetes
```

- Verify that the download works and has the expected version:

  ```
  kubeadm version
  ```

- Verify the upgrade plan:

  ```
  kubeadm upgrade plan
  ```

  This command checks that your cluster can be upgraded, and fetches the versions you can upgrade to. It also shows a table with the component config version states.

Note: `kubeadm upgrade` also automatically renews the certificates that it manages on this node. To opt-out of certificate renewal the flag `--certificate-renewal=false` can be used. For more information see the certificate management guide.
Note: If `kubeadm upgrade plan` shows any component configs that require manual upgrade, users must provide a config file with replacement configs to `kubeadm upgrade apply` via the `--config` command line flag. Failing to do so will cause `kubeadm upgrade apply` to exit with an error and not perform an upgrade.

- Choose a version to upgrade to, and run the appropriate command. For example:

  ```
  # replace x with the patch version you picked for this upgrade
  sudo kubeadm upgrade apply v1.23.x
  ```

  Once the command finishes you should see:

  ```
  [upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.23.x". Enjoy!

  [upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading your kubelets if you haven't already done so.
  ```

- Manually upgrade your CNI provider plugin.

  Your Container Network Interface (CNI) provider may have its own upgrade instructions to follow. Check the addons page to find your CNI provider and see whether additional upgrade steps are required.

This step is not required on additional control plane nodes if the CNI provider runs as a DaemonSet.

For the other control plane nodes

Same as the first control plane node but use:

```
sudo kubeadm upgrade node
```

instead of:

```
sudo kubeadm upgrade apply
```

Also calling `kubeadm upgrade plan` and upgrading the CNI provider plugin is no longer needed.

Drain the node

- Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

  ```
  # replace <node-to-drain> with the name of your node you are
  draining
  kubectl drain <node-to-drain> --ignore-daemonsets
  ```

Upgrade kubelet and kubectl

- Upgrade the kubelet and kubectl:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.23.x-00
kubectl=1.23.x-00 && \
apt-mark hold kubelet kubectl
```

```
# replace x in 1.23.x-0 with the latest patch version
yum install -y kubelet-1.23.x-0 kubectl-1.23.x-0 --
disableexcludes=kubernetes
```

- Restart the kubelet:

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

Uncordon the node

- Bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

Upgrade worker nodes

The upgrade procedure on worker nodes should be executed one node at a time or few nodes at a time, without compromising the minimum required capacity for running your workloads.

Upgrade kubeadm

- Upgrade kubeadm:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.23.x-00 && \
apt-mark hold kubeadm
```

```
# replace x in 1.23.x-0 with the latest patch version
yum install -y kubeadm-1.23.x-0 --disableexcludes=kubernetes
```

Call "kubeadm upgrade"

- For worker nodes this upgrades the local kubelet configuration:

  ```
  sudo kubeadm upgrade node
  ```

Drain the node

- Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

  ```
  # replace <node-to-drain> with the name of your node you are draining
  kubectl drain <node-to-drain> --ignore-daemonsets
  ```

Upgrade kubelet and kubectl

- Upgrade the kubelet and kubectl:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora

```
# replace x in 1.23.x-00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.23.x-00
kubectl=1.23.x-00 && \
apt-mark hold kubelet kubectl
```

```
# replace x in 1.23.x-0 with the latest patch version
yum install -y kubelet-1.23.x-0 kubectl-1.23.x-0 --
disableexcludes=kubernetes
```

- Restart the kubelet:

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

Uncordon the node

- Bring the node back online by marking it schedulable:

  ```
  # replace <node-to-drain> with the name of your node
  kubectl uncordon <node-to-drain>
  ```

Verify the status of the cluster

After the kubelet is upgraded on all nodes verify that all nodes are available again by running the following command from anywhere kubectl can access the cluster:

```
kubectl get nodes
```

The `STATUS` column should show `Ready` for all your nodes, and the version number should be updated.

Recovering from a failure state

If `kubeadm upgrade` fails and does not roll back, for example because of an unexpected shutdown during execution, you can run `kubeadm upgrade` again. This command is idempotent and eventually makes sure that the actual state is the desired state you declare.

To recover from a bad state, you can also run `kubeadm upgrade apply --force` without changing the version that your cluster is running.

During upgrade kubeadm writes the following backup folders under `/etc/kubernetes/tmp`:

- `kubeadm-backup-etcd-<date>-<time>`
- `kubeadm-backup-manifests-<date>-<time>`

`kubeadm-backup-etcd` contains a backup of the local etcd member data for this control plane Node. In case of an etcd upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in `/var/lib/etcd`. In case external etcd is used this backup folder will be empty.

`kubeadm-backup-manifests` contains a backup of the static Pod manifest files for this control plane Node. In case of a upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in `/etc/kubernetes/manifests`. If for some reason there is no difference between a pre-upgrade and post-upgrade manifest file for a certain component, a backup file for it will not be written.

How it works

`kubeadm upgrade apply` does the following:

- Checks that your cluster is in an upgradeable state:
  - The API server is reachable
  - All nodes are in the `Ready` state
  - The control plane is healthy
- Enforces the version skew policies.
- Makes sure the control plane images are available or available to pull to the machine.
- Generates replacements and/or uses user supplied overwrites if component configs require version upgrades.
- Upgrades the control plane components or rollbacks if any of them fails to come up.
- Applies the new `CoreDNS` and `kube-proxy` manifests and makes sure that all necessary RBAC rules are created.
- Creates new certificate and key files of the API server and backs up old files if they're about to expire in 180 days.

`kubeadm upgrade node` does the following on additional control plane nodes:

- Fetches the kubeadm `ClusterConfiguration` from the cluster.
- Optionally backups the kube-apiserver certificate.
- Upgrades the static Pod manifests for the control plane components.
- Upgrades the kubelet configuration for this node.

`kubeadm upgrade node` does the following on worker nodes:

- Fetches the kubeadm `ClusterConfiguration` from the cluster.
- Upgrades the kubelet configuration for this node.

Adding Windows nodes

FEATURE STATE: `Kubernetes v1.18 [beta]`

You can use Kubernetes to run a mixture of Linux and Windows nodes, so you can mix Pods that run on Linux on with Pods that run on Windows. This page shows how to register Windows nodes to your cluster.

Before you begin

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

- Obtain a Windows Server 2019 license (or higher) in order to configure the Windows node that hosts Windows containers. If you are using VXLAN/Overlay networking you must have also have KB4489899 installed.
- A Linux-based Kubernetes kubeadm cluster in which you have access to the control plane (see Creating a single control-plane cluster with kubeadm).

Objectives

- Register a Windows node to the cluster
- Configure networking so Pods and Services on Linux and Windows can communicate with each other

Getting Started: Adding a Windows Node to Your Cluster

Networking Configuration

Once you have a Linux-based Kubernetes control-plane node you are ready to choose a networking solution. This guide illustrates using Flannel in VXLAN mode for simplicity.

Configuring Flannel

1. Prepare Kubernetes control plane for Flannel

   Some minor preparation is recommended on the Kubernetes control plane in our cluster. It is recommended to enable bridged IPv4 traffic to iptables chains when using Flannel. The following command must be run on all Linux nodes:

   ```
   sudo sysctl net.bridge.bridge-nf-call-iptables=1
   ```

2. Download & configure Flannel for Linux

   Download the most recent Flannel manifest:

   ```
   wget https://raw.githubusercontent.com/coreos/flannel/master/
   Documentation/kube-flannel.yml
   ```

   Modify the `net-conf.json` section of the flannel manifest in order to set the VNI to 4096 and the Port to 4789. It should look as follows:

```
net-conf.json: |
    {
      "Network": "10.244.0.0/16",
      "Backend": {
        "Type": "vxlan",
        "VNI": 4096,
        "Port": 4789
      }
    }
```

Note: The VNI must be set to 4096 and port 4789 for Flannel on Linux to interoperate with Flannel on Windows. See the VXLAN documentation. for an explanation of these fields.
Note: To use L2Bridge/Host-gateway mode instead change the value of `Type` to `"host-gw"` and omit `VNI` and `Port`.

3. Apply the Flannel manifest and validate

   Let's apply the Flannel configuration:

   ```
   kubectl apply -f kube-flannel.yml
   ```

   After a few minutes, you should see all the pods as running if the Flannel pod network was deployed.

   ```
   kubectl get pods -n kube-system
   ```

   The output should include the Linux flannel DaemonSet as running:

   ```
   NAMESPACE     NAME
   READY        STATUS    RESTARTS   AGE
   ...
   kube-system   kube-flannel-ds-54954
   1/1          Running   0          1m
   ```

4. Add Windows Flannel and kube-proxy DaemonSets

   Now you can add Windows-compatible versions of Flannel and kube-proxy. In order to ensure that you get a compatible version of kube-proxy, you'll need to substitute the tag of the image. The following example shows usage for Kubernetes v1.23.0, but you should adjust the version for your own deployment.

   ```
   curl -L https://github.com/kubernetes-sigs/sig-windows-tools/
   releases/latest/download/kube-proxy.yml | sed 's/VERSION/
   v1.23.0/g' | kubectl apply -f -
   kubectl apply -f https://github.com/kubernetes-sigs/sig-
   windows-tools/releases/latest/download/flannel-overlay.yml
   ```

   Note: If you're using host-gateway use https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel-host-gw.yml instead
   Note:

If you're using a different interface rather than Ethernet (i.e. "Ethernet0 2") on the Windows nodes, you have to modify the line:

```
wins cli process run --path /k/flannel/setup.exe --args "--mode=overlay --interface=Ethernet"
```

in the `flannel-host-gw.yml` or `flannel-overlay.yml` file and specify your interface accordingly.

```
# Example
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel-overlay.yml | sed 's/Ethernet/Ethernet0 2/g' | kubectl apply -f -
```

Joining a Windows worker node

Note: All code snippets in Windows sections are to be run in a PowerShell environment with elevated permissions (Administrator) on the Windows worker node.

- Docker EE
- CRI-containerD

Install Docker EE

Install the `Containers` feature

```
Install-WindowsFeature -Name containers
```

Install Docker Instructions to do so are available at Install Docker Engine - Enterprise on Windows Servers.

Install wins, kubelet, and kubeadm

```
curl.exe -LO https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master/kubeadm/scripts/PrepareNode.ps1
.\PrepareNode.ps1 -KubernetesVersion v1.23.0
```

Run `kubeadm` to join the node

Use the command that was given to you when you ran `kubeadm init` on a control plane host. If you no longer have this command, or the token has expired, you can run `kubeadm token create --print-join-command` (on a control plane host) to generate a new token and join command.

Install containerD

```
curl.exe -LO https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/Install-Containerd.ps1
.\Install-Containerd.ps1
```

Note:

To install a specific version of containerD specify the version with -ContainerDVersion.

```
# Example
.\Install-Containerd.ps1 -ContainerDVersion 1.4.1
```

Note:

If you're using a different interface rather than Ethernet (i.e. "Ethernet0 2") on the Windows nodes, specify the name with `-netAdapterName`.

```
# Example
.\Install-Containerd.ps1 -netAdapterName "Ethernet0 2"
```

Install wins, kubelet, and kubeadm

```
curl.exe -LO https://raw.githubusercontent.com/kubernetes-sigs/sig-windows-tools/master/kubeadm/scripts/PrepareNode.ps1
.\PrepareNode.ps1 -KubernetesVersion v1.23.0 -ContainerRuntime containerD
```

Run `kubeadm` to join the node

Use the command that was given to you when you ran `kubeadm init` on a control plane host. If you no longer have this command, or the token has expired, you can run `kubeadm token create --print-join-command` (on a control plane host) to generate a new token and join command.

Note: If using **CRI-containerD** add `--cri-socket "npipe:////./pipe/containerd-containerd"` to the kubeadm call

Verifying your installation

You should now be able to view the Windows node in your cluster by running:

```
kubectl get nodes -o wide
```

If your new node is in the `NotReady` state it is likely because the flannel image is still downloading. You can check the progress as before by checking on the flannel pods in the `kube-system` namespace:

```
kubectl -n kube-system get pods -l app=flannel
```

Once the flannel Pod is running, your node should enter the `Ready` state and then be available to handle workloads.

What's next

- Upgrading Windows kubeadm nodes

Upgrading Windows nodes

FEATURE STATE: `Kubernetes v1.18 [beta]`

This page explains how to upgrade a Windows node created with kubeadm.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

- Familiarize yourself with the process for upgrading the rest of your kubeadm cluster. You will want to upgrade the control plane nodes before upgrading your Windows nodes.

Upgrading worker nodes

Upgrade kubeadm

1. From the Windows node, upgrade kubeadm:

   ```
   # replace v1.23.0 with your desired version
   curl.exe -Lo C:\k\kubeadm.exe https://dl.k8s.io//bin/windows/
   amd64/kubeadm.exe
   ```

Drain the node

1. From a machine with access to the Kubernetes API, prepare the node for maintenance by marking it unschedulable and evicting the workloads:

   ```
   # replace <node-to-drain> with the name of your node you are
   draining
   kubectl drain <node-to-drain> --ignore-daemonsets
   ```

   You should see output similar to this:

   ```
   node/ip-172-31-85-18 cordoned
   node/ip-172-31-85-18 drained
   ```

Upgrade the kubelet configuration

1. From the Windows node, call the following command to sync new kubelet configuration:

   ```
   kubeadm upgrade node
   ```

Upgrade kubelet

1. From the Windows node, upgrade and restart the kubelet:

   ```
   stop-service kubelet
   curl.exe -Lo C:\k\kubelet.exe https://dl.k8s.io//bin/windows/
   amd64/kubelet.exe
   restart-service kubelet
   ```

Uncordon the node

1. From a machine with access to the Kubernetes API, bring the node back online by marking it schedulable:

   ```
   # replace <node-to-drain> with the name of your node
   kubectl uncordon <node-to-drain>
   ```

Upgrade kube-proxy

1. From a machine with access to the Kubernetes API, run the following, again replacing v1.23.0 with your desired version:

   ```
   curl -L https://github.com/kubernetes-sigs/sig-windows-tools/
   releases/latest/download/kube-proxy.yml | sed 's/VERSION/
   v1.23.0/g' | kubectl apply -f -
   ```

Migrating from dockershim

This section presents information you need to know when migrating from dockershim to other container runtimes.

Since the announcement of dockershim deprecation in Kubernetes 1.20, there were questions on how this will affect various workloads and Kubernetes installations. Our Dockershim Removal FAQ is there to help you to understand the problem better.

Dockershim will be removed from Kubernetes following the release of v1.24. If you use Docker via dockershim as your container runtime, and wish to upgrade to v1.24, it is recommended that you either migrate to another runtime or find an alternative means to obtain Docker Engine support. If you're not sure whether you are using Docker, find out what container runtime is used on a node.

Your cluster might have more than one kind of node, although this is not a common configuration.

These tasks will help you to migrate:

- Check whether Dockershim deprecation affects you
- Migrating from dockershim
- Migrating telemetry and security agents from dockershim

What's next

- Check out container runtimes to understand your options for a container runtime.
- There is a GitHub issue to track discussion about the deprecation and removal of dockershim.
- If you found a defect or other technical concern relating to migrating away from dockershim, you can report an issue to the Kubernetes project.

Changing the Container Runtime on a Node from Docker Engine to containerd

This task outlines the steps needed to update your container runtime to containerd from Docker. It is applicable for cluster operators running Kubernetes 1.23 or earlier. Also this covers an example scenario for migrating from dockershim to containerd and alternative container runtimes can be picked from this page.

Before you begin

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Install containerd. For more information see containerd's installation documentation and for specific prerequisite follow the containerd guide.

Drain the node

```
kubectl drain <node-to-drain> --ignore-daemonsets
```

Replace `<node-to-drain>` with the name of your node you are draining.

Stop the Docker daemon

```
systemctl stop kubelet
systemctl disable docker.service --now
```

Install Containerd

Follow the guide for detailed steps to install containerd.

- Linux
- Windows (PowerShell)

1. Install the `containerd.io` package from the official Docker repositories. Instructions for setting up the Docker repository for your respective Linux distribution and installing the `containerd.io` package can be found at Install Docker Engine.

2. Configure containerd:

   ```
   sudo mkdir -p /etc/containerd
   containerd config default | sudo tee /etc/containerd/
   config.toml
   ```

3. Restart containerd:

   ```
   sudo systemctl restart containerd
   ```

Start a Powershell session, set `$Version` to the desired version (ex: `$Version="1.4.3"`), and then run the following commands:

1. Download containerd:

   ```
   curl.exe -L https://github.com/containerd/containerd/
   releases/download/v$Version/containerd-$Version-windows-
   amd64.tar.gz -o containerd-windows-amd64.tar.gz
   tar.exe xvf .\containerd-windows-amd64.tar.gz
   ```

2. Extract and configure:

   ```
   Copy-Item -Path ".\bin\" -Destination "$Env:ProgramFiles\cont
   ainerd" -Recurse -Force
   cd $Env:ProgramFiles\containerd\
   .\containerd.exe config default | Out-File config.toml -
   Encoding ascii

   # Review the configuration. Depending on setup you may want
   to adjust:
   # - the sandbox_image (Kubernetes pause image)
   # - cni bin_dir and conf_dir locations
   Get-Content config.toml

   # (Optional - but highly recommended) Exclude containerd
   ```

```
from Windows Defender Scans
Add-MpPreference -ExclusionProcess "$Env:ProgramFiles\contain
erd\containerd.exe"
```

3. Start containerd:

```
.\containerd.exe --register-service
Start-Service containerd
```

Configure the kubelet to use containerd as its container runtime

Edit the file `/var/lib/kubelet/kubeadm-flags.env` and add the containerd runtime to the flags. `--container-runtime=remote` and `--container-runtime-endpoint=unix:///run/containerd/containerd.sock"`.

Users using kubeadm should be aware that the `kubeadm` tool stores the CRI socket for each host as an annotation in the Node object for that host. To change it you can execute the following command on a machine that has the kubeadm `/etc/kubernetes/admin.conf` file.

```
kubectl edit no <node-name>
```

This will start a text editor where you can edit the Node object. To choose a text editor you can set the `KUBE_EDITOR` environment variable.

- Change the value of `kubeadm.alpha.kubernetes.io/cri-socket` from `/var/run/dockershim.sock` to the CRI socket path of your choice (for example `unix:///run/containerd/containerd.sock`).

  Note that new CRI socket paths must be prefixed with `unix://` ideally.

- Save the changes in the text editor, which will update the Node object.

Restart the kubelet

```
systemctl start kubelet
```

Verify that the node is healthy

Run `kubectl get nodes -o wide` and containerd appears as the runtime for the node we just changed.

Remove Docker Engine

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Finally if everything goes well, remove Docker.

- CentOS
- Debian
- Fedora
- Ubuntu

```
sudo yum remove docker-ce docker-ce-cli
```

```
sudo apt-get purge docker-ce docker-ce-cli
```

```
sudo dnf remove docker-ce docker-ce-cli
```

```
sudo apt-get purge docker-ce docker-ce-cli
```

Migrate Docker Engine nodes from dockershim to cri-dockerd

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

This page shows you how to migrate your Docker Engine nodes to use `cri-dockerd` instead of dockershim. Follow these steps if your clusters run Kubernetes 1.23 or earlier and you want to continue using Docker Engine after you upgrade to Kubernetes 1.24 and later, or if you just want to move off the dockershim component.

What is cri-dockerd?

In Kubernetes 1.23 and earlier, Docker Engine used a component called the dockershim to interact with Kubernetes system components such as the kubelet. The dockershim component is deprecated and will be removed in Kubernetes 1.24. A third-party replacement, `cri-dockerd`, is available. The `cri-dockerd` adapter lets you use Docker Engine through the Container Runtime Interface.

Note: If you already use `cri-dockerd`, you aren't affected by the dockershim removal. Before you begin, Check whether your nodes use the dockershim.

If you want to migrate to `cri-dockerd` so that you can continue using Docker Engine as your container runtime, you should do the following for each affected node:

1. Install `cri-dockerd`.
2. Cordon and drain the node.
3. Configure the kubelet to use `cri-dockerd`.
4. Restart the kubelet.

5. Verify that the node is healthy.

Test the migration on non-critical nodes first.

You should perform the following steps for each node that you want to migrate to `cri-dockerd`.

Before you begin

- `cri-dockerd` installed and started on each node.
- A network plugin.

Cordon and drain the node

1. Cordon the node to stop new Pods scheduling on it:

   ```
   kubectl cordon <NODE_NAME>
   ```

   Replace `<NODE_NAME>` with the name of the node.

2. Drain the node to safely evict running Pods:

   ```
   kubectl drain <NODE_NAME> \
       --ignore-daemonsets
   ```

Configure the kubelet to use cri-dockerd

The following steps apply to clusters set up using the kubeadm tool. If you use a different tool, you should modify the kubelet using the configuration instructions for that tool.

1. Open `/var/lib/kubelet/kubeadm-flags.env` on each affected node.
2. Modify the `--container-runtime-endpoint` flag to `unix:///var/run/cri-dockerd.sock`.

The kubeadm tool stores the node's socket as an annotation on the `Node` object in the control plane. To modify this socket for each affected node:

1. Edit the YAML representation of the `Node` object:

   ```
   KUBECONFIG=/path/to/admin.conf kubectl edit no <NODE_NAME>
   ```

   Replace the following:

   - `/path/to/admin.conf`: the path to the kubectl configuration file, `admin.conf`.
   - `<NODE_NAME>`: the name of the node you want to modify.

2. Change `kubeadm.alpha.kubernetes.io/cri-socket` from `/var/run/dockershim.sock` to `unix:///var/run/cri-dockerd.sock`.

3. Save the changes. The `Node` object is updated on save.

Restart the kubelet

```
systemctl restart kubelet
```

Verify that the node is healthy

To check whether the node uses the `cri-dockerd` endpoint, follow the instructions in Find out which runtime you use. The `--container-runtime-endpoint` flag for the kubelet should be `unix:///var/run/cri-dockerd.sock`.

Uncordon the node

Uncordon the node to let Pods schedule on it:

```
kubectl uncordon <NODE_NAME>
```

What's next

- Read the dockershim removal FAQ.
- Learn how to migrate from Docker Engine with dockershim to containerd.

Find Out What Container Runtime is Used on a Node

This page outlines steps to find out what container runtime the nodes in your cluster use.

Depending on the way you run your cluster, the container runtime for the nodes may have been pre-configured or you need to configure it. If you're using a managed Kubernetes service, there might be vendor-specific ways to check what container runtime is configured for the nodes. The method described on this page should work whenever the execution of `kubectl` is allowed.

Before you begin

Install and configure `kubectl`. See Install Tools section for details.

Find out the container runtime used on a Node

Use `kubectl` to fetch and show node information:

```
kubectl get nodes -o wide
```

The output is similar to the following. The column `CONTAINER-RUNTIME` outputs the runtime and its version.

For Docker Engine, the output is similar to this:

```
NAME         STATUS   VERSION    CONTAINER-RUNTIME
node-1       Ready    v1.16.15   docker://19.3.1
node-2       Ready    v1.16.15   docker://19.3.1
node-3       Ready    v1.16.15   docker://19.3.1
```

If your runtime shows as Docker Engine, you still might not be affected by the removal of dockershim in Kubernetes 1.24. Check the runtime endpoint to see if you use dockershim. If you don't use dockershim, you aren't affected.

For containerd, the output is similar to this:

```
NAME         STATUS   VERSION   CONTAINER-RUNTIME
node-1       Ready    v1.19.6   containerd://1.4.1
node-2       Ready    v1.19.6   containerd://1.4.1
node-3       Ready    v1.19.6   containerd://1.4.1
```

Find out more information about container runtimes on Container Runtimes page.

Find out what container runtime endpoint you use

The container runtime talks to the kubelet over a Unix socket using the CRI protocol, which is based on the gRPC framework. The kubelet acts as a client, and the runtime acts as the server. In some cases, you might find it useful to know which socket your nodes use. For example, with the removal of dockershim in Kubernetes 1.24 and later, you might want to know whether you use Docker Engine with dockershim.

Note: If you currently use Docker Engine in your nodes with `cri-dockerd`, you aren't affected by the dockershim removal.

You can check which socket you use by checking the kubelet configuration on your nodes.

1. Read the starting commands for the kubelet process:

   ```
   tr \\0 ' ' < /proc/"$(pgrep kubelet)"/cmdline
   ```

   If you don't have `tr` or `pgrep`, check the command line for the kubelet process manually.

2. In the output, look for the `--container-runtime` flag and the `--container-runtime-endpoint` flag.
    - If your nodes use Kubernetes v1.23 and earlier and these flags aren't present or if the `--container-runtime` flag is not `remote`, you use the dockershim socket with Docker Engine.
    - If the `--container-runtime-endpoint` flag is present, check the socket name to find out which runtime you use. For example, `unix:///run/containerd/containerd.sock` is the containerd endpoint.

If you use Docker Engine with the dockershim, migrate to a different runtime, or, if you want to continue using Docker Engine in v1.24 and later, migrate to a CRI-compatible adapter like `cri-dockerd`.

Check whether Dockershim deprecation affects you

The `dockershim` component of Kubernetes allows to use Docker as a Kubernetes's container runtime. Kubernetes' built-in `dockershim` component was deprecated in release v1.20.

This page explains how your cluster could be using Docker as a container runtime, provides details on the role that `dockershim` plays when in use, and shows steps you can take to check whether any workloads could be affected by `dockershim` deprecation.

Finding if your app has a dependencies on Docker

If you are using Docker for building your application containers, you can still run these containers on any container runtime. This use of Docker does not count as a dependency on Docker as a container runtime.

When alternative container runtime is used, executing Docker commands may either not work or yield unexpected output. This is how you can find whether you have a dependency on Docker:

1. Make sure no privileged Pods execute Docker commands (like `docker ps`), restart the Docker service (commands such as `systemctl restart docker.service`), or modify Docker-specific files such as `/etc/docker/daemon.json`.
2. Check for any private registries or image mirror settings in the Docker configuration file (like `/etc/docker/daemon.json`). Those typically need to be reconfigured for another container runtime.
3. Check that scripts and apps running on nodes outside of your Kubernetes infrastructure do not execute Docker commands. It might be:
    - SSH to nodes to troubleshoot;

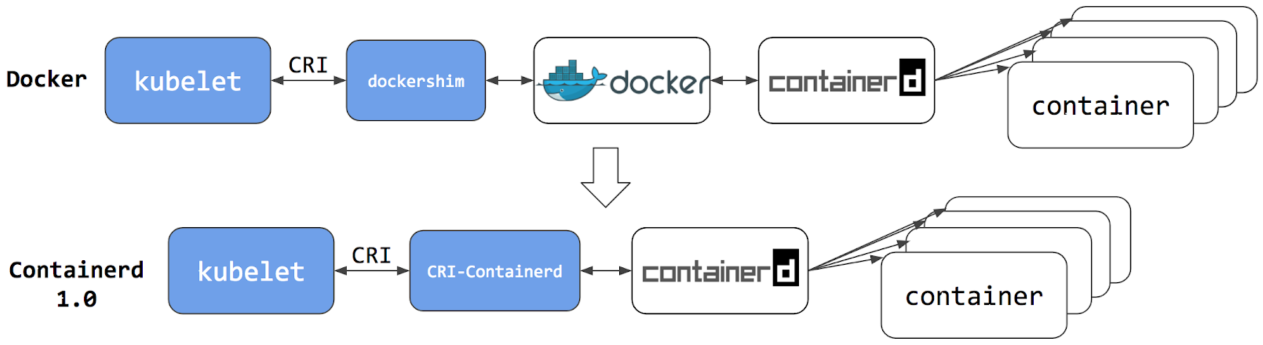
    - Node startup scripts;
    - Monitoring and security agents installed on nodes directly.
4. Third-party tools that perform above mentioned privileged operations. See Migrating telemetry and security agents from dockershim for more information.
5. Make sure there is no indirect dependencies on dockershim behavior. This is an edge case and unlikely to affect your application. Some tooling may be configured to react to Docker-specific behaviors, for example, raise alert on specific metrics or search for a specific log message as part of troubleshooting instructions. If you have such tooling configured, test the behavior on test cluster before migration.

Dependency on Docker explained

A container runtime is software that can execute the containers that make up a Kubernetes pod. Kubernetes is responsible for orchestration and scheduling of Pods; on each node, the kubelet uses the container runtime interface as an abstraction so that you can use any compatible container runtime.

In its earliest releases, Kubernetes offered compatibility with one container runtime: Docker. Later in the Kubernetes project's history, cluster operators wanted to adopt additional container runtimes. The CRI was designed to allow this kind of flexibility - and the kubelet began supporting CRI. However, because Docker existed before the CRI specification was invented, the Kubernetes project created an adapter component, `dockershim`. The dockershim adapter allows the kubelet to interact with Docker as if Docker were a CRI compatible runtime.

You can read about it in Kubernetes Containerd integration goes GA blog post.



Switching to Containerd as a container runtime eliminates the middleman. All the same containers can be run by container runtimes like Containerd as before. But now, since containers schedule directly with the container runtime, they are not visible to Docker. So any Docker tooling or fancy UI you might have used before to check on these containers is no longer available.

You cannot get container information using `docker ps` or `docker inspect` commands. As you cannot list containers, you cannot get logs, stop containers, or execute something inside container using `docker exec`.

Note: If you're running workloads via Kubernetes, the best way to stop a container is through the Kubernetes API rather than directly through the container runtime (this advice applies for all container runtimes, not only Docker).

You can still pull images or build them using `docker build` command. But images built or pulled by Docker would not be visible to container runtime and Kubernetes. They needed to be pushed to some registry to allow them to be used by Kubernetes.

What's next

- Read Migrating from dockershim to understand your next steps
- Read the dockershim deprecation FAQ article for more information.

Migrating telemetry and security agents from dockershim

Kubernetes' support for direct integration with Docker Engine is deprecated, and will be removed. Most apps do not have a direct dependency on runtime hosting containers. However, there are still a lot of telemetry and monitoring agents that has a dependency on docker to collect containers metadata, logs and metrics. This document aggregates information on how to detect these dependencies and links on how to migrate these agents to use generic tools or alternative runtimes.

Telemetry and security agents

Within a Kubernetes cluster there are a few different ways to run telemetry or security agents. Some agents have a direct dependency on Docker Engine when they as DaemonSets or directly on nodes.

Why do some telemetry agents communicate with Docker Engine?

Historically, Kubernetes was written to work specifically with Docker Engine. Kubernetes took care of networking and scheduling, relying on Docker Engine for launching and running containers (within Pods) on a node. Some information that is relevant to telemetry, such as a pod name, is only available from Kubernetes components. Other data, such as container metrics, is not the responsibility of the container runtime. Early telemetry agents needed to query the container runtime **and** Kubernetes to report an accurate picture. Over time, Kubernetes gained the ability to support multiple runtimes, and now supports any runtime that is compatible with the container runtime interface.

Some telemetry agents rely specifically on Docker Engine tooling. For example, an agent might run a command such as `docker ps` or `docker top` to list containers and processes or `docker logs` to receive streamed logs. If nodes in your existing cluster use Docker Engine, and you switch to a different container runtime, these commands will not work any longer.

Identify DaemonSets that depend on Docker Engine

If a pod wants to make calls to the `dockerd` running on the node, the pod must either:

- mount the filesystem containing the Docker daemon's privileged socket, as a volume; or
- mount the specific path of the Docker daemon's privileged socket directly, also as a volume.

For example: on COS images, Docker exposes its Unix domain socket at `/var/run/docker.sock` This means that the pod spec will include a `hostPath` volume mount of `/var/run/docker.sock`.

Here's a sample shell script to find Pods that have a mount directly mapping the Docker socket. This script outputs the namespace and name of the pod. You can remove the `grep '/var/run/docker.sock'` to review other mounts.

```
kubectl get pods --all-namespaces \
-o=jsonpath='{range .items[*]}{"\n"}{.metadata.namespace}{":\t"}
{.metadata.name}{":\t"}{range .spec.volumes[*]}{.hostPath.path}
{", "}{end}{end}' \
| sort \
| grep '/var/run/docker.sock'
```

Note: There are alternative ways for a pod to access Docker on the host. For instance, the parent directory `/var/run` may be mounted instead of the full path (like in this example). The script above only detects the most common uses.

Detecting Docker dependency from node agents

In case your cluster nodes are customized and install additional security and telemetry agents on the node, make sure to check with the vendor of the agent whether it has dependency on Docker.

Telemetry and security agent vendors

We keep the work in progress version of migration instructions for various telemetry and security agent vendors in Google doc. Please contact the vendor to get up to date instructions for migrating from dockershim.

Certificates

When using client certificate authentication, you can generate certificates manually through `easyrsa`, `openssl` or `cfssl`.

easyrsa

easyrsa can manually generate certificates for your cluster.

1. Download, unpack, and initialize the patched version of easyrsa3.

   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/
   easy-rsa/easy-rsa.tar.gz
   tar xzf easy-rsa.tar.gz
   cd easy-rsa-master/easyrsa3
   ./easyrsa init-pki
   ```

2. Generate a new certificate authority (CA). `--batch` sets automatic mode; `--req-cn` specifies the Common Name (CN) for the CA's new root certificate.

   ```
   ./easyrsa --batch "--req-cn=${MASTER_IP}@`date +%s`" build-
   ca nopass
   ```

3. Generate server certificate and key. The argument `--subject-alt-name` sets the possible IPs and DNS names the API server will be accessed with. The `MASTER_CLUSTER_IP` is usually the first IP from the service CIDR that is specified as the `--service-cluster-ip-range` argument for both the API server and the controller manager component. The argument `--days` is used to set the number of days after which the certificate expires. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

   ```
   ./easyrsa --subject-alt-name="IP:${MASTER_IP},"\
   "IP:${MASTER_CLUSTER_IP},"\
   "DNS:kubernetes,"\
   "DNS:kubernetes.default,"\
   "DNS:kubernetes.default.svc,"\
   "DNS:kubernetes.default.svc.cluster,"\
   "DNS:kubernetes.default.svc.cluster.local" \
   --days=10000 \
   build-server-full server nopass
   ```

4. Copy `pki/ca.crt`, `pki/issued/server.crt`, and `pki/private/server.key` to your directory.

5. Fill in and add the following parameters into the API server start parameters:

   ```
   --client-ca-file=/yourdirectory/ca.crt
   --tls-cert-file=/yourdirectory/server.crt
   --tls-private-key-file=/yourdirectory/server.key
   ```

openssl

openssl can manually generate certificates for your cluster.

1. Generate a ca.key with 2048bit:

   ```
   openssl genrsa -out ca.key 2048
   ```

2. According to the ca.key generate a ca.crt (use -days to set the certificate effective time):

   ```
   openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER_IP}" -days 10000 -out ca.crt
   ```

3. Generate a server.key with 2048bit:

   ```
   openssl genrsa -out server.key 2048
   ```

4. Create a config file for generating a Certificate Signing Request (CSR). Be sure to substitute the values marked with angle brackets (e.g. `<MASTER_IP>`) with real values before saving this to a file (e.g. `csr.conf`). Note that the value for `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

   ```
   [ req ]
   default_bits = 2048
   prompt = no
   default_md = sha256
   req_extensions = req_ext
   distinguished_name = dn

   [ dn ]
   C = <country>
   ST = <state>
   L = <city>
   O = <organization>
   OU = <organization unit>
   CN = <MASTER_IP>

   [ req_ext ]
   subjectAltName = @alt_names

   [ alt_names ]
   DNS.1 = kubernetes
   DNS.2 = kubernetes.default
   DNS.3 = kubernetes.default.svc
   DNS.4 = kubernetes.default.svc.cluster
   DNS.5 = kubernetes.default.svc.cluster.local
   IP.1 = <MASTER_IP>
   IP.2 = <MASTER_CLUSTER_IP>
   ```

```
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alt_names
```

5. Generate the certificate signing request based on the config file:

```
openssl req -new -key server.key -out server.csr -config
csr.conf
```

6. Generate the server certificate using the ca.key, ca.crt and server.csr:

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 10000 \
-extensions v3_ext -extfile csr.conf
```

7. View the certificate signing request:

```
openssl req  -noout -text -in ./server.csr
```

8. View the certificate:

```
openssl x509  -noout -text -in ./server.crt
```

Finally, add the same parameters into the API server start parameters.

cfssl

cfssl is another tool for certificate generation.

1. Download, unpack and prepare the command line tools as shown below. Note that you may need to adapt the sample commands based on the hardware architecture and cfssl version you are using.

```
curl -L https://github.com/cloudflare/cfssl/releases/
download/v1.5.0/cfssl_1.5.0_linux_amd64 -o cfssl
chmod +x cfssl
curl -L https://github.com/cloudflare/cfssl/releases/
download/v1.5.0/cfssljson_1.5.0_linux_amd64 -o cfssljson
chmod +x cfssljson
curl -L https://github.com/cloudflare/cfssl/releases/
download/v1.5.0/cfssl-certinfo_1.5.0_linux_amd64 -o cfssl-
certinfo
chmod +x cfssl-certinfo
```

2. Create a directory to hold the artifacts and initialize cfssl:

```
mkdir cert
cd cert
../cfssl print-defaults config > config.json
../cfssl print-defaults csr > csr.json
```

3. Create a JSON config file for generating the CA file, for example, `ca-config.json`:

   ```
   {
     "signing": {
       "default": {
         "expiry": "8760h"
       },
       "profiles": {
         "kubernetes": {
           "usages": [
             "signing",
             "key encipherment",
             "server auth",
             "client auth"
           ],
           "expiry": "8760h"
         }
       }
     }
   }
   ```

4. Create a JSON config file for CA certificate signing request (CSR), for example, `ca-csr.json`. Be sure to replace the values marked with angle brackets with real values you want to use.

   ```
   {
     "CN": "kubernetes",
     "key": {
       "algo": "rsa",
       "size": 2048
     },
     "names":[{
       "C": "<country>",
       "ST": "<state>",
       "L": "<city>",
       "O": "<organization>",
       "OU": "<organization unit>"
     }]
   }
   ```

5. Generate CA key (`ca-key.pem`) and certificate (`ca.pem`):

   ```
   ../cfssl gencert -initca ca-csr.json | ../cfssljson -bare ca
   ```

6. Create a JSON config file for generating keys and certificates for the API server, for example, `server-csr.json`. Be sure to replace the values in angle brackets with real values you want to use. The `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

```
{
  "CN": "kubernetes",
  "hosts": [
    "127.0.0.1",
    "<MASTER_IP>",
    "<MASTER_CLUSTER_IP>",
    "kubernetes",
    "kubernetes.default",
    "kubernetes.default.svc",
    "kubernetes.default.svc.cluster",
    "kubernetes.default.svc.cluster.local"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [{
    "C": "<country>",
    "ST": "<state>",
    "L": "<city>",
    "O": "<organization>",
    "OU": "<organization unit>"
  }]
}
```

7. Generate the key and certificate for the API server, which are by default saved into file `server-key.pem` and `server.pem` respectively:

```
../cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
--config=ca-config.json -profile=kubernetes \
server-csr.json | ../cfssljson -bare server
```

Distributing Self-Signed CA Certificate

A client node may refuse to recognize a self-signed CA certificate as valid. For a non-production deployment, or for a deployment that runs behind a company firewall, you can distribute a self-signed CA certificate to all clients and refresh the local list for valid certificates.

On each client, perform the following operations:

```
sudo cp ca.crt /usr/local/share/ca-certificates/kubernetes.crt
sudo update-ca-certificates
```

```
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....
done.
```

Certificates API

You can use the `certificates.k8s.io` API to provision x509 certificates to use for authentication as documented here.

Manage Memory, CPU, and API Resources

Configure Default Memory Requests and Limits for a Namespace

Define a default memory resource limit for a namespace, so that every new Pod in that namespace has a memory resource limit configured.

Configure Default CPU Requests and Limits for a Namespace

Define a default CPU resource limits for a namespace, so that every new Pod in that namespace has a CPU resource limit configured.

Configure Minimum and Maximum Memory Constraints for a Namespace

Define a range of valid memory resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

Configure Minimum and Maximum CPU Constraints for a Namespace

Define a range of valid CPU resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

Configure Memory and CPU Quotas for a Namespace

Define overall memory and CPU resource limits for a namespace.

Configure a Pod Quota for a Namespace

Restrict how many Pods you can create within a namespace.

Configure Default Memory Requests and Limits for a Namespace

Define a default memory resource limit for a namespace, so that every new Pod in that namespace has a memory resource limit configured.

This page shows how to configure default memory requests and limits for a namespace.

A Kubernetes cluster can be divided into namespaces. Once you have a namespace that has a default memory limit, and you then try to create a Pod with a container that does not specify its own memory limit, then the control plane assigns the default memory limit to that container.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

Each node in your cluster must have at least 2 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 a manifest for an example LimitRange. The manifest specifies a default memory request and a default memory limit.

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/memory-defaults.yaml --namespace=default-mem-example
```

Now if you create a Pod in the default-mem-example namespace, and any container within that Pod does not specify its own values for memory request and memory limit, then the control plane applies default values: a memory request of 256MiB and a memory limit of 512MiB.

Here's an example manifest for a Pod that has one container. The container does not specify a memory request and limit.

admin/resource/memory-defaults-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo
spec:
  containers:
  - name: default-mem-demo-ctr
    image: nginx
```

Create the Pod.

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod.yaml --namespace=default-mem-example
```

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.

```
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 a manifest for a Pod that has one container. The container specifies a memory limit, but not a request:

admin/resource/memory-defaults-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-2
spec:
  containers:
  - name: default-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1Gi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-
defaults-pod-2.yaml --namespace=default-mem-example
```

View detailed information about the Pod:

```
kubectl get pod default-mem-demo-2 --output=yaml --namespace=defa
ult-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 a manifest for a Pod that has one container. The container specifies a memory request, but not a limit:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod-3.yaml --namespace=default-mem-example
```

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 manifest. The container is limited to use no more than 512MiB of memory, which matches 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 memory resource quota configured, 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:

- For every Pod that runs in the namespace, the Pod and each of its containers must have a memory limit. (If you specify a memory limit for every container in a Pod, Kubernetes can infer the Pod-level memory limit by adding up the limits for its containers).
- Memory limits apply a resource reservation on the node where the Pod in question is scheduled. The total amount of memory reserved for all Pods in the namespace must not exceed a specified limit.

- The total amount of memory actually used by all Pods in the namespace must also not exceed a specified limit.

When you add a LimitRange:

If any Pod in that namespace that includes a container does not specify its own memory limit, the control plane applies the default memory limit to that container, and the Pod can be allowed to run in a namespace that is restricted by a memory ResourceQuota.

Clean up

Delete your namespace:

```
kubectl delete namespace default-mem-example
```

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

Define a default CPU resource limits for a namespace, so that every new Pod in that namespace has a CPU resource limit configured.

This page shows how to configure default CPU requests and limits for a namespace.

A Kubernetes cluster can be divided into namespaces. If you create a Pod within a namespace that has a default CPU limit, and any container in that Pod does not specify its own CPU limit, then the control plane assigns the default CPU limit to that container.

Kubernetes assigns a default CPU request, but only under certain conditions that are explained later in this page.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

If you're not already familiar with what Kubernetes means by 1.0 CPU, read meaning of 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 default-cpu-example
```

Create a LimitRange and a Pod

Here's a manifest for an example LimitRange. The manifest specifies a default CPU request and a default CPU limit.

admin/resource/cpu-defaults.yaml

```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 apply -f https://k8s.io/examples/admin/resource/cpu-
defaults.yaml --namespace=default-cpu-example
```

Now if you create a Pod in the default-cpu-example namespace, and any container in that Pod does not specify its own values for CPU request and CPU limit, then the control plane applies default values: a CPU request of 0.5 and a default CPU limit of 1.

Here's a manifest for a Pod that has one container. The container does not specify a CPU request and limit.

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/cpu-
defaults-pod.yaml --namespace=default-cpu-example
```

View the Pod's specification:

```
kubectl get pod default-cpu-demo --output=yaml --namespace=defaul
t-cpu-example
```

The output shows that the Pod's only container has a CPU request of 500m `cpu` (which you can read as "500 millicpu"), 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 a manifest for a Pod that has one container. The container specifies a CPU limit, but not a request:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/cpu-
defaults-pod-2.yaml --namespace=default-cpu-example
```

View the specification of the Pod that you created:

```
kubectl get pod default-cpu-demo-2 --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 an example manifest for a Pod that has one container. The container specifies a CPU request, but not a limit:

admin/resource/cpu-defaults-pod-3.yaml

```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 apply -f https://k8s.io/examples/admin/resource/cpu-
defaults-pod-3.yaml --namespace=default-cpu-example
```

View the specification of the Pod that you created:

```
kubectl get pod default-cpu-demo-3 --output=yaml --
namespace=default-cpu-example
```

The output shows that the container's CPU request is set to the value you specified at the time you created the Pod (in other words: it matches the manifest). However, the same container's CPU limit is set to 1 `cpu`, which is the default CPU limit for that namespace.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 750m
```

Motivation for default CPU limits and requests

If your namespace has a CPU resource quota configured, it is helpful to have a default value in place for CPU limit. Here are two of the restrictions that a CPU resource quota imposes on a namespace:

- For every Pod that runs in the namespace, each of its containers must have a CPU limit.
- CPU limits apply a resource reservation on the node where the Pod in question is scheduled. The total amount of CPU that is reserved for use by all Pods in the namespace must not exceed a specified limit.

When you add a LimitRange:

If any Pod in that namespace that includes a container does not specify its own CPU limit, the control plane applies the default CPU limit to that container, and the Pod can be allowed to run in a namespace that is restricted by a CPU ResourceQuota.

Clean up

Delete your namespace:

```
kubectl delete namespace default-cpu-example
```

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

Define a range of valid memory resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

Each node in your cluster must have at least 1 GiB of memory available for Pods.

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 an example manifest for a LimitRange:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/memory-
constraints.yaml --namespace=constraints-mem-example
```

View detailed information about the LimitRange:

```
kubectl get limitrange mem-min-max-demo-lr --namespace=constraint
s-mem-example --output=yaml
```

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 you define a Pod within the constraints-mem-example namespace, Kubernetes performs these steps:

- If any container in that Pod does not specify its own memory request and limit, assign the default memory request and limit to that container.
- Verify that every container in that Pod requests at least 500 MiB of memory.
- Verify that every container in that Pod requests no more than 1024 MiB (1 GiB) of memory.

Here's a manifest for a Pod that has one container. Within the Pod spec, the sole container 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.

admin/resource/memory-constraints-pod.yaml

```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 apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod.yaml --namespace=constraints-mem-example
```

Verify that the Pod is running and that its container is healthy:

```
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=co
nstraints-mem-example
```

The output shows that the container within that Pod has a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the constraints imposed by the LimitRange for this namespace:

```
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 a manifest for a Pod that has one container. The container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB.

admin/resource/memory-constraints-pod-2.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 apply -f https://k8s.io/examples/admin/resource/memory-
constraints-pod-2.yaml --namespace=constraints-mem-example
```

The output shows that the Pod does not get created, because it defines a container that requests more memory than is allowed:

```
Error from server (Forbidden): error when creating "examples/
admin/resource/memory-constraints-pod-2.yaml":
pods "constraints-mem-demo-2" is forbidden: maximum memory usage
per Container is 1Gi, but limit is 1536Mi.
```

Attempt to create a Pod that does not meet the minimum memory request

Here's a manifest for a Pod that has one container. That container specifies a memory request of 100 MiB and a memory limit of 800 MiB.

admin/resource/memory-constraints-pod-3.yaml

```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 apply -f https://k8s.io/examples/admin/resource/memory-
constraints-pod-3.yaml --namespace=constraints-mem-example
```

The output shows that the Pod does not get created, because it defines a container that requests less memory than the enforced minimum:

```
Error from server (Forbidden): error when creating "examples/
admin/resource/memory-constraints-pod-3.yaml":
pods "constraints-mem-demo-3" is forbidden: minimum memory usage
per Container is 500Mi, but request is 100Mi.
```

Create a Pod that does not specify any memory request or limit

Here's a manifest for a Pod that has one container. The container does not specify a memory request, and it does not specify a memory limit.

admin/resource/memory-constraints-pod-4.yaml

```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 apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-4.yaml --namespace=constraints-mem-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo-4 --namespace=constraints-mem-example --output=yaml
```

The output shows that the Pod's only container has a memory request of 1 GiB and a memory limit of 1 GiB. How did that container get those values?

```
resources:
  limits:
    memory: 1Gi
  requests:
    memory: 1Gi
```

Because your Pod did not define any memory request and limit for that container, the cluster applied a default memory request and limit from the LimitRange.

This means that the definition of that Pod shows those values. You can check it using `kubectl describe`:

```
# Look for the "Requests:" section of the output
kubectl describe pod constraints-mem-demo-4 --namespace=constraints-mem-example
```

At this point, your Pod 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 GiB of memory. You do not want to accept any Pod that requests more than 2 GiB 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 GiB of memory, but you want development workloads to be limited to 512 MiB. 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

Define a range of valid CPU resource limits for a namespace, so that every new Pod in that namespace falls within the range you configure.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

Your cluster must have at least 1.0 CPU available for use to run the task examples. See meaning of CPU to learn what Kubernetes means by "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 an example manifest for a LimitRange:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/cpu-
constraints.yaml --namespace=constraints-cpu-example
```

View detailed information about the LimitRange:

```
kubectl get limitrange cpu-min-max-demo-lr --output=yaml --
namespace=constraints-cpu-example
```

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 you create a Pod in the constraints-cpu-example namespace (or some other client of the Kubernetes API creates an equivalent Pod), Kubernetes performs these steps:

- If any container in that Pod does not specify its own CPU request and limit, the control plane assigns the default CPU request and limit to that container.
- Verify that every container in that Pod specifies a CPU request that is greater than or equal to 200 millicpu.
- Verify that every container in that Pod specifies a CPU limit that is less than or equal to 800 millicpu.

Note: When creating a `LimitRange` object, you can specify limits on huge-pages or GPUs as well. However, when both `default` and `defaultRequest` are specified on these resources, the two values must be the same.

Here's a manifest 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.

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod.yaml --namespace=constraints-cpu-example
```

Verify that the Pod is running and that its container is healthy:

```
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-example
```

The output shows that the Pod's only 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 a manifest for a Pod that has one container. The container specifies a CPU request of 500 millicpu and a cpu limit of 1.5 cpu.

admin/resource/cpu-constraints-pod-2.yaml

```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 apply -f https://k8s.io/examples/admin/resource/cpu-
constraints-pod-2.yaml --namespace=constraints-cpu-example
```

The output shows that the Pod does not get created, because it defines an unacceptable container. That container is not acceptable because it specifies a CPU limit that is too large:

```
Error from server (Forbidden): error when creating "examples/
admin/resource/cpu-constraints-pod-2.yaml":
pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage
per Container is 800m, but limit is 1500m.
```

Attempt to create a Pod that does not meet the minimum CPU request

Here's a manifest for a Pod that has one container. The container specifies a CPU request of 100 millicpu and a CPU limit of 800 millicpu.

admin/resource/cpu-constraints-pod-3.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-3
spec:
  containers:
  - name: constraints-cpu-demo-3-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-3.yaml --namespace=constraints-cpu-example
```

The output shows that the Pod does not get created, because it defines an unacceptable container. That container is not acceptable because it specifies a CPU request that is lower than the enforced minimum:

```
Error from server (Forbidden): error when creating "examples/admin/resource/cpu-constraints-pod-3.yaml":
pods "constraints-cpu-demo-3" is forbidden: minimum cpu usage per Container is 200m, but request is 100m.
```

Create a Pod that does not specify any CPU request or limit

Here's a manifest for a Pod that has one container. The container does not specify a CPU request, nor does it specify a CPU limit.

admin/resource/cpu-constraints-pod-4.yaml

```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 apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-4.yaml --namespace=constraints-cpu-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo-4 --namespace=constraints-cpu-example --output=yaml
```

The output shows that the Pod's single container has a CPU request of 800 millicpu and a CPU limit of 800 millicpu. How did that container get those values?

```
resources:
  limits:
    cpu: 800m
  requests:
    cpu: 800m
```

Because that container did not specify its own CPU request and limit, the control plane applied the default CPU request and limit from the LimitRange for this namespace.

At this point, your Pod might be running or it might not be running. Recall that a prerequisite for this task is that your cluster must have at least 1 CPU available for use. 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

Define overall memory and CPU resource limits for a namespace.

This page shows how to set quotas for the total amount memory and CPU that can be used by all Pods 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

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 a manifest for an example ResourceQuota:

admin/resource/quota-mem-cpu.yaml

```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 apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu.yaml --namespace=quota-mem-cpu-example
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=yaml
```

The ResourceQuota places these requirements on the quota-mem-cpu-example namespace:

- For every Pod in the namespace, each container must have a memory request, memory limit, cpu request, and cpu limit.
- The memory request total for all Pods in that namespace must not exceed 1 GiB.
- The memory limit total for all Pods in that namespace must not exceed 2 GiB.
- The CPU request total for all Pods in that namespace must not exceed 1 cpu.
- The CPU limit total for all Pods in that namespace must not exceed 2 cpu.

See meaning of CPU to learn what Kubernetes means by "1 CPU".

Create a Pod

Here is a manifest for an example Pod:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-pod.yaml --namespace=quota-mem-cpu-example
```

Verify that the Pod is running and that its (only) container is healthy:

```
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=yaml
```

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
```

If you have the `jq` tool, you can also query (using JSONPath) for just the `use d` values, **and** pretty-print that that of the output. For example:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-
example -o jsonpath='{ .status.used }' | jq .
```

Attempt to create a second Pod

Here is a manifest for a second Pod:

admin/resource/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 manifest, 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 apply -f https://k8s.io/examples/admin/resource/quota-
mem-cpu-pod-2.yaml --namespace=quota-mem-cpu-example
```

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 "examples/
admin/resource/quota-mem-cpu-pod-2.yaml":
pods "quota-mem-cpu-demo-2" is forbidden: exceeded quota: mem-
cpu-demo,
requested: requests.memory=700Mi,used: requests.memory=600Mi,
limited: requests.memory=1Gi
```

Discussion

As you have seen in this exercise, you can use a ResourceQuota to restrict the memory request total for all Pods running in a namespace. You can also restrict the totals for memory limit, cpu request, and cpu limit.

Instead of managing total resource use within a namespace, you might want to restrict individual Pods, or the containers in those Pods. To achieve that kind of limiting, 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

Restrict how many Pods you can create within 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 must have access to create namespaces in your cluster.

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 an example manifest for a ResourceQuota:

admin/resource/quota-pod.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
```

```yaml
  hard:
    pods: "2"
```

Create the ResourceQuota:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod.yaml --namespace=quota-pod-example
```

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.

```yaml
spec:
  hard:
    pods: "2"
status:
  hard:
    pods: "2"
  used:
    pods: "0"
```

Here is an example manifest for a Deployment:

admin/resource/quota-pod-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-quota-demo
spec:
  selector:
    matchLabels:
      purpose: quota-demo
  replicas: 3
  template:
    metadata:
      labels:
        purpose: quota-demo
    spec:
      containers:
      - name: pod-quota-demo
        image: nginx
```

In that manifest, `replicas: 3` tells Kubernetes to attempt to create three new Pods, all running the same application.

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod-deployment.yaml --namespace=quota-pod-example
```

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 you defined earlier:

```
spec:
  ...
  replicas: 3
...
status:
  availableReplicas: 2
...
lastUpdateTime: 2021-04-02T20:57:05Z
    message: 'unable to create pods: pods "pod-quota-demo-1650323038-" is forbidden:
      exceeded quota: pod-demo, requested: pods=1, used: pods=2, limited: pods=2'
```

Choice of resource

In this task you have defined a ResourceQuota that limited the total number of Pods, but you could also limit the total number of other kinds of object. For example, you might decide to limit how many CronJobs that can live in a single namespace.

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

Install a Network Policy Provider

Use Antrea for NetworkPolicy

Use Calico for NetworkPolicy

Use Cilium for NetworkPolicy

Use Kube-router for NetworkPolicy

Romana for NetworkPolicy

Weave Net for NetworkPolicy

Use Antrea for NetworkPolicy

This page shows how to install and use Antrea CNI plugin on Kubernetes. For background on Project Antrea, read the Introduction to Antrea.

Before you begin

You need to have a Kubernetes cluster. Follow the kubeadm getting started guide to bootstrap one.

Deploying Antrea with kubeadm

Follow Getting Started guide to deploy Antrea for kubeadm.

What's next

Once your cluster is running, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

Use Calico for NetworkPolicy

This page shows a couple of quick ways to create a Calico cluster on Kubernetes.

Before you begin

Decide whether you want to deploy a cloud or local cluster.

Creating a Calico cluster with Google Kubernetes Engine (GKE)

Prerequisite: gcloud.

1. To launch a GKE cluster with Calico, include the `--enable-network-policy` flag.

   Syntax

   ```
   gcloud container clusters create [CLUSTER_NAME] --enable-network-policy
   ```

   Example

   ```
   gcloud container clusters create my-calico-cluster --enable-network-policy
   ```

2. To verify the deployment, use the following command.

   ```
   kubectl get pods --namespace=kube-system
   ```

   The Calico pods begin with `calico`. Check to make sure each one has a status of `Running`.

Creating a local Calico cluster with kubeadm

To get a local single-host Calico cluster in fifteen minutes using kubeadm, refer to the Calico Quickstart.

What's next

Once your cluster is running, you can follow the Declare Network Policy 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

To start minikube, minimal version required is >= v1.5.2, run the with the following arguments:

```
minikube version
```

```
minikube version: v1.5.2
```

```
minikube start --network-plugin=cni
```

For minikube you can install Cilium using its CLI tool. Cilium will automatically detect the cluster configuration and will install the appropriate components for a successful installation:

```
curl -LO https://github.com/cilium/cilium-cli/releases/latest/
download/cilium-linux-amd64.tar.gz
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin
rm cilium-linux-amd64.tar.gz
cilium install
```

```
ðŸ”® Auto-detected Kubernetes kind: minikube
âœ¨ Running "minikube" validation checks
âœ… Detected minikube version "1.20.0"
â„¹ï¸�  Cilium version not set, using default version "v1.10.0"
ðŸ”® Auto-detected cluster name: minikube
ðŸ”® Auto-detected IPAM mode: cluster-pool
ðŸ”® Auto-detected datapath mode: tunnel
ðŸ”‘ Generating CA...
2021/05/27 02:54:44 [INFO] generate received request
2021/05/27 02:54:44 [INFO] received CSR
```

```
2021/05/27 02:54:44 [INFO] generating key: ecdsa-256
2021/05/27 02:54:44 [INFO] encoded CSR
2021/05/27 02:54:44 [INFO] signed certificate with serial number
487137649188566744011364712294827030212305386\42
ðŸ”‘ Generating certificates for Hubble...
2021/05/27 02:54:44 [INFO] generate received request
2021/05/27 02:54:44 [INFO] received CSR
2021/05/27 02:54:44 [INFO] generating key: ecdsa-256
2021/05/27 02:54:44 [INFO] encoded CSR
2021/05/27 02:54:44 [INFO] signed certificate with serial number
351410973402578431008638918842156061333327957\4
ðŸš€ Creating Service accounts...
ðŸš€ Creating Cluster roles...
ðŸš€ Creating ConfigMap...
ðŸš€ Creating Agent DaemonSet...
ðŸš€ Creating Operator Deployment...
âŒ› Waiting for Cilium to be installed...
```

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 Kubernetes Installation 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 -l k8s-app=cilium
```

You'll see a list of Pods similar to this:

```
NAME            READY   STATUS    RESTARTS   AGE
cilium-kkdhz    1/1     Running   0          3m23s
...
```

A `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.

What's next

Once your cluster is running, you can follow the Declare Network Policy 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 Declare Network Policy 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 kubeadm.

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 you have installed Romana, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

Weave Net for NetworkPolicy

This page shows how to use Weave Net for NetworkPolicy.

Before you begin

You need to have a Kubernetes cluster. Follow the kubeadm getting started guide to bootstrap one.

Install the 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.

Test the installation

Verify that the weave works.

Enter the following command:

```
kubectl get pods -n kube-system -o wide
```

The output is similar to this:

```
NAME                                    READY     STATUS
RESTARTS   AGE       IP              NODE
weave-net-1t1qg                            2/2       Running
0          9d        192.168.2.10    worknode3
weave-net-231d7                            2/2       Running
1          7d        10.2.0.17       worknodegpu
weave-net-7nmwt                            2/2       Running
3          9d        192.168.2.131   masternode
weave-net-pmw8w                            2/2       Running
0          9d        192.168.2.216   worknode2
```

Each Node has a weave Pod, and all Pods are `Running` and `2/2 READY`. (2/2 means that each Pod has `weave` and `weave-npc`.)

What's next

Once you have installed the Weave Net addon, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy. If you have any question, contact us at #weave-community on Slack or Weave User Group.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Accessing the Kubernetes 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 API server. 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 API server:

1. Run kubectl in proxy mode (recommended). This method is recommended, since it uses the stored apiserver location and verifies the identity of the API server 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 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 API server 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/
```

The output is similar to this:

```
{
  "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 API server, like this:

Using `grep/cut` approach:

```
# Check all possible clusters, as your .KUBECONFIG may have
multiple contexts:
```

```
kubectl config view -o jsonpath='{"Cluster name\tServer\n"}
{range .clusters[*]}{.name}{"\t"}{.cluster.server}{"\n"}{end}'

# Select name of cluster you want to interact with from above
output:
export CLUSTER_NAME="some_server_name"

# Point to the API server referring the cluster name
APISERVER=$(kubectl config view -o jsonpath="{.clusters[?
(@.name==\"$CLUSTER_NAME\")].cluster.server}")

# Create a secret to hold a token for the default service account
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: default-token
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF

# Wait for the token controller to populate the secret with a
token:
while ! kubectl describe secret default-token | grep -E '^token'
>/dev/null; do
  echo "waiting for token..." >&2
  sleep 1
done

# Get the token value
TOKEN=$(kubectl get secret default-token -o jsonpath='{.data.toke
n}' | base64 --decode)

# Explore the API with TOKEN
curl -X GET $APISERVER/api --header "Authorization: Bearer
$TOKEN" --insecure
```

The output is similar to this:

```
{
  "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 API server does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. [Controlling Access to the Kubernetes API]() describes how you can configure this as a cluster administrator.

Programmatic access to the API

Kubernetes officially supports client libraries for [Go](), [Python](), [Java](), [dotnet](), [Javascript](), and [Haskell](). There are other client libraries that are provided and maintained by their authors, not the Kubernetes team. See [client libraries]() for accessing the API from other languages and how they authenticate.

Go client

- To get the library, run the following command: `go get k8s.io/client-go@kubernetes-<kubernetes-version-number>` See [https://github.com/kubernetes/client-go/releases]() to see which versions are supported.
- Write an application atop of the client-go clients.

Note: client-go defines its own API objects, so if needed, import API definitions from client-go rather than from the main repository. For example, `import "k8s.io/client-go/kubernetes"` is correct.

The Go client can use the same [kubeconfig file]() as the kubectl CLI does to locate and authenticate to the API server. See this [example]():

```
package main

import (
  "context"
  "fmt"
  "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/clientcmd"
)

func main() {
  // uses the current context in kubeconfig
  // path-to-kubeconfig -- for example, /root/.kube/config
  config, _ := clientcmd.BuildConfigFromFlags("", "<path-to-
kubeconfig>")
  // creates the clientset
  clientset, _ := kubernetes.NewForConfig(config)
  // access the API to list pods
  pods, _ := clientset.CoreV1().Pods("").List(context.TODO(),
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, see Accessing the API from within a Pod.

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 API server. 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))
```

Java client

To install the Java Client, run:

```
# Clone java library
git clone --recursive https://github.com/kubernetes-client/java

# Installing project artifacts, POM etc:
cd java
mvn install
```

See https://github.com/kubernetes-client/java/releases to see which versions are supported.

The Java client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
package io.kubernetes.client.examples;

import io.kubernetes.client.ApiClient;
import io.kubernetes.client.ApiException;
import io.kubernetes.client.Configuration;
import io.kubernetes.client.apis.CoreV1Api;
import io.kubernetes.client.models.V1Pod;
import io.kubernetes.client.models.V1PodList;
import io.kubernetes.client.util.ClientBuilder;
import io.kubernetes.client.util.KubeConfig;
import java.io.FileReader;
import java.io.IOException;

/**
 * A simple example of how to use the Java API from an
application outside a kubernetes cluster
 *
 * <p>Easiest way to run this: mvn exec:java
 * -
Dexec.mainClass="io.kubernetes.client.examples.KubeConfigFileClie
ntExample"
 *
 */
public class KubeConfigFileClientExample {
  public static void main(String[] args) throws IOException,
ApiException {

    // file path to your KubeConfig
    String kubeConfigPath = "~/.kube/config";

    // loading the out-of-cluster config, a kubeconfig from file-
system
    ApiClient client =
        ClientBuilder.kubeconfig(KubeConfig.loadKubeConfig(new
FileReader(kubeConfigPath))).build();

    // set the global default api-client to the in-cluster one
from above
    Configuration.setDefaultApiClient(client);

    // the CoreV1Api loads default api-client from global
configuration.
    CoreV1Api api = new CoreV1Api();

    // invokes the CoreV1Api client
    V1PodList list = api.listPodForAllNamespaces(null, null,
null, null, null, null, null, null, null);
    System.out.println("Listing all pods: ");
    for (V1Pod item : list.getItems()) {
      System.out.println(item.getMetadata().getName());
    }
  }
}
```

dotnet client

To use dotnet client, run the following command: `dotnet add package KubernetesClient --version 1.6.1` See dotnet Client Library page for more installation options. See https://github.com/kubernetes-client/csharp/releases to see which versions are supported.

The dotnet client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
using System;
using k8s;

namespace simple
{
    internal class PodList
    {
        private static void Main(string[] args)
        {
            var config =
KubernetesClientConfiguration.BuildDefaultConfig();
            IKubernetes client = new Kubernetes(config);
            Console.WriteLine("Starting Request!");

            var list = client.ListNamespacedPod("default");
            foreach (var item in list.Items)
            {
                Console.WriteLine(item.Metadata.Name);
            }
            if (list.Items.Count == 0)
            {
                Console.WriteLine("Empty!");
            }
        }
    }
}
```

JavaScript client

To install JavaScript client, run the following command: `npm install @kubernetes/client-node`. See https://github.com/kubernetes-client/javascript/releases to see which versions are supported.

The JavaScript client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
const k8s = require('@kubernetes/client-node');

const kc = new k8s.KubeConfig();
kc.loadFromDefault();

const k8sApi = kc.makeApiClient(k8s.CoreV1Api);

k8sApi.listNamespacedPod('default').then((res) => {
    console.log(res.body);
});
```

Haskell client

See https://github.com/kubernetes-client/haskell/releases to see which versions are supported.

The Haskell client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
exampleWithKubeConfig :: IO ()
exampleWithKubeConfig = do
    oidcCache <- atomically $ newTVar $ Map.fromList []
    (mgr, kcfg) <- mkKubeClientConfig oidcCache $ KubeConfigFile
"/path/to/kubeconfig"
    dispatchMime
            mgr
            kcfg
            (CoreV1.listPodForAllNamespaces (Accept MimeJSON))
        >>= print
```

What's next

- Accessing the Kubernetes API from a Pod

Advertise Extended Resources for a Node

This page shows how to specify extended resources for a Node. Extended resources allow cluster administrators to advertise node-level resources that would otherwise be unknown to Kubernetes.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Get the names of your Nodes

```
kubectl get nodes
```

Choose one of your Nodes to use for this exercise.

Advertise a new extended resource on one of your Nodes

To advertise a new extended 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/example.com~1dongle",
    "value": "4"
  }
]
```

Note that Kubernetes does not need to know what a dongle is or what a dongle is for. The preceding PATCH request 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/
example.com~1dongle", "value": "4"}]' \
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": {
  "cpu": "2",
  "memory": "2049008Ki",
  "example.com/dongle": "4",
```

Describe your Node:

```
kubectl describe node <your-node-name>
```

Once again, the output shows the dongle resource:

```
Capacity:
 cpu:  2
 memory:  2049008Ki
 example.com/dongle:  4
```

Now, application developers can create Pods that request a certain number of dongles. See Assign Extended Resources to a Container.

Discussion

Extended 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.

Extended resources are opaque to Kubernetes; Kubernetes does not know anything about what they are. Kubernetes knows only that a Node has a certain number of them. Extended resources 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 example.com/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 example.com/special-storage.

```
Capacity:
 ...
 example.com/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 example.com/special-storage.

```
Capacity:
 ...
 example.com/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/example.com~1dongle",
  }
]
```

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/
example.com~1dongle"}]' \
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
```

(you should not see any output)

What's next

For application developers

- Assign Extended 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

Autoscale the DNS Service in a Cluster

This page shows how to enable and configure autoscaling of the DNS service in your 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

- This guide assumes your nodes use the AMD64 or Intel 64 CPU architecture.

- Make sure Kubernetes DNS is enabled.

Determine 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                      READY   UP-TO-DATE   AVAILABLE   AGE
...
dns-autoscaler            1/1     1            1           ...
...
```

If you see "dns-autoscaler" in the output, DNS horizontal autoscaling is already enabled, and you can skip to Tuning autoscaling parameters.

Get the name of your DNS Deployment

List the DNS deployments in your cluster in the kube-system namespace:

```
kubectl get deployment -l k8s-app=kube-dns --namespace=kube-
system
```

The output is similar to this:

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
...
coredns   2/2     2            2           ...
...
```

If you don't see a Deployment for DNS services, you can also look for it by name:

```
kubectl get deployment --namespace=kube-system
```

and look for a deployment named `coredns` or `kube-dns`.

Your scale target is

```
Deployment/<your-deployment-name>
```

where `<your-deployment-name>` is the name of your DNS Deployment. For example, if the name of your Deployment for DNS is coredns, your scale target is Deployment/coredns.

Note: CoreDNS is the default DNS service for Kubernetes. CoreDNS sets the label `k8s-app=kube-dns` so that it can work in clusters that originally used kube-dns.

Enable DNS horizontal autoscaling

In this section, you create a new 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:

admin/dns/dns-horizontal-autoscaler.yaml

```yaml
kind: ServiceAccount
apiVersion: v1
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
rules:
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["list", "watch"]
  - apiGroups: [""]
```

```yaml
    resources: ["replicationcontrollers/scale"]
    verbs: ["get", "update"]
  - apiGroups: ["apps"]
    resources: ["deployments/scale", "replicasets/scale"]
    verbs: ["get", "update"]
# Remove the configmaps rule once below issue is fixed:
# kubernetes-incubator/cluster-proportional-autoscaler#16
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["get", "create"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
subjects:
  - kind: ServiceAccount
    name: kube-dns-autoscaler
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-dns-autoscaler
  apiGroup: rbac.authorization.k8s.io

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: kube-dns-autoscaler
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: kube-dns-autoscaler
  template:
    metadata:
      labels:
        k8s-app: kube-dns-autoscaler
    spec:
      priorityClassName: system-cluster-critical
      securityContext:
        seccompProfile:
          type: RuntimeDefault
        supplementalGroups: [ 65534 ]
        fsGroup: 65534
      nodeSelector:
        kubernetes.io/os: linux
      containers:
      - name: autoscaler
```

```
        image: k8s.gcr.io/cpa/cluster-proportional-autoscaler:
1.8.4
        resources:
            requests:
                cpu: "20m"
                memory: "10Mi"
        command:
          - /cluster-proportional-autoscaler
          - --namespace=kube-system
          - --configmap=kube-dns-autoscaler
          # Should keep target in sync with cluster/addons/dns/
kube-dns.yaml.base
          - --target=<SCALE_TARGET>
          # When cluster is using large nodes(with more cores),
"coresPerReplica" should dominate.
          # If using small nodes, "nodesPerReplica" should
dominate.
          - --default-params={"linear":{"coresPerReplica":
256,"nodesPerReplica":
16,"preventSinglePointFailure":true,"includeUnschedulableNodes":t
rue}}
          - --logtostderr=true
          - --v=2
      tolerations:
      - key: "CriticalAddonsOnly"
        operator: "Exists"
      serviceAccountName: kube-dns-autoscaler
```

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 apply -f dns-horizontal-autoscaler.yaml
```

The output of a successful command is:

```
deployment.apps/dns-autoscaler created
```

DNS horizontal autoscaling is now enabled.

Tune DNS autoscaling parameters

Verify that the dns-autoscaler [ConfigMap](https://) exists:

```
kubectl get configmap --namespace=kube-system
```

The output is similar to this:

```
NAME                  DATA      AGE
...
```

```
dns-autoscaler        1         ...
...
```

Modify the data in the ConfigMap:

```
kubectl edit configmap 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 is calculated using this equation:

```
replicas = max( ceil( cores Ã— 1/coresPerReplica ) , ceil( nodes
Ã— 1/nodesPerReplica ) )
```

Note that the values of both `coresPerReplica` and `nodesPerReplica` are floats.

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 tuning DNS horizontal autoscaling. Which option to use depends on different conditions.

Option 1: Scale down the dns-autoscaler deployment to 0 replicas

This option works for all situations. Enter this command:

```
kubectl scale deployment --replicas=0 dns-autoscaler --
namespace=kube-system
```

The output is:

```
deployment.apps/dns-autoscaler scaled
```

Verify that the replica count is zero:

```
kubectl get rs --namespace=kube-system
```

The output displays 0 in the DESIRED and CURRENT columns:

```
NAME                                 DESIRED   CURRENT   READY
AGE
```

```
...
dns-autoscaler-6b59789fc8          0        0
0       ...
...
```

Option 2: Delete the dns-autoscaler deployment

This option works if dns-autoscaler is under your own control, which means no one will re-create it:

```
kubectl delete deployment dns-autoscaler --namespace=kube-system
```

The output is:

```
deployment.apps "dns-autoscaler" deleted
```

Option 3: Delete the dns-autoscaler manifest file from the master node

This option works if dns-autoscaler is under control of the (deprecated) Addon Manager, 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 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 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*.

What's next

- Read about Guaranteed Scheduling For Critical Add-On Pods.
- Learn more about the implementation of cluster-proportional-autoscaler.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

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                 PROVISIONER               AGE
   standard (default)   kubernetes.io/gce-pd      1d
   gold                 kubernetes.io/gce-pd      1d
   ```

   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 standard -p '{"metadata":
   {"annotations":{"storageclass.kubernetes.io/is-default-
   class":"false"}}}'
   ```

   where `standard` is the name of your chosen StorageClass.

3. Mark a StorageClass as default:

   Similar to the previous step, you need to add/set the annotation `storageclass.kubernetes.io/is-default-class=true`.

   ```
   kubectl patch storageclass gold -p '{"metadata":
   {"annotations":{"storageclass.kubernetes.io/is-default-
   class":"true"}}}'
   ```

   Please note that at most one StorageClass can be marked as default. If two or more of them are marked as default, a `PersistentVolumeClaim` without `storageClassName` explicitly specified cannot be created.

4. Verify that your chosen StorageClass is default:

   ```
   kubectl get storageclass
   ```

   The output is similar to this:

   ```
   NAME             PROVISIONER               AGE
   standard         kubernetes.io/gce-pd      1d
   gold (default)   kubernetes.io/gce-pd      1d
   ```

What's next

- Learn more about PersistentVolumes.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 PersistentVolumeClaim. 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 will 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   RECLAIMPOLICY   STATUS    CLAIM
STORAGECLASS     REASON    AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Delete          Bound     default/claim1
manual                     10s
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Delete          Bound     default/claim2
manual                     6s
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Delete          Bound     default/claim3
manual                     3s
```

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":"Retain"}}'
```

where `<your-pv-name>` is the name of your chosen PersistentVolume.

Note:

On Windows, you must *double* quote any JSONPath template that contains spaces (not single quote as shown above for bash). This in turn means that you must use a single quote or escaped double quote around any literals in the template. For example:

```
kubectl patch pv <your-pv-name> -p "{\"spec\":{\"persistentVo
lumeReclaimPolicy\":\"Retain\"}}"
```

3. Verify that your chosen PersistentVolume has the right policy:

```
kubectl get pv
```

The output is similar to this:

```
NAME                                       CAPACITY
ACCESSMODES   RECLAIMPOLICY   STATUS    CLAIM
STORAGECLASS     REASON    AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Delete          Bound     default/claim1
manual                     40s
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Delete          Bound     default/claim2
manual                     36s
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi
RWO           Retain          Bound     default/claim3
manual                     33s
```

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.

References

- PersistentVolume
  - Pay attention to the `.spec.persistentVolumeReclaimPolicy` field of PersistentVolume.
- PersistentVolumeClaim

Cloud Controller Manager Administration

FEATURE STATE: `Kubernetes v1.11 [beta]`

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 satisfies 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.

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 availability: 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.

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 unschedulable. The taint is important since the scheduler may require cloud specific information about nodes such as their 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 retrieve 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.

The 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
- 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 repositories maintained by cloud vendors or by SIGs.

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:

admin/cloud/ccm-example.yaml

```yaml
# This is an example of how to setup cloud-controller-manager as a Daemonset in your cluster.
# It assumes that your masters can run pods and has the role node-role.kubernetes.io/master
# Note that this Daemonset will not work straight out of the box for your cloud, this is
# meant to be a guideline.

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
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: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
```

```yaml
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 k8s.gcr.io/cloud-
controller-manager
        # this can be replaced with any other image for out-of-
tree providers
        image: k8s.gcr.io/cloud-controller-manager:v1.8.0
        command:
        - /usr/local/bin/cloud-controller-manager
        - --cloud-provider=[YOUR_CLOUD_PROVIDER]  # Add your own
cloud provider here!
        - --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

The cloud-controller-manager queries your cloud provider's APIs to retrieve information for all nodes. For very large clusters, consider possible bottlenecks 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. Unfortunately, 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. TLS bootstrapping 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 initialized 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.

What's next

To build and develop your own cloud controller manager, read Developing Cloud Controller Manager.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

admin/resource/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 apply -f https://k8s.io/examples/admin/resource/quota-
objects.yaml --namespace=quota-object-example
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota object-quota-demo --namespace=quota-
object-example --output=yaml
```

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:

admin/resource/quota-objects-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-quota-demo
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects-pvc.yaml --namespace=quota-object-example
```

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:

admin/resource/quota-objects-pvc-2.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-quota-demo-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
```

Attempt to create the second PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-
objects-pvc-2.yaml --namespace=quota-object-example
```

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

Control CPU Management Policies on the Node

FEATURE STATE: `Kubernetes v1.12 [beta]`

Kubernetes keeps many aspects of how pods execute on nodes abstracted from the user. This is by design. Â 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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

CPU Management Policies

By default, the kubelet uses CFS quota to enforce pod CPU limits. Â 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 policy is set with the `--cpu-manager-policy` kubelet flag or the `cpuManagerPolicy` field in KubeletConfiguration. There are two supported policies:

- `none`: the default policy.
- `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`.

The behavior of the static policy can be fine-tuned using the `--cpu-manager-policy-options` flag. The flag takes a comma-separated list of `key=value` policy options. This feature can be disabled completely using the `CPUManagerPolicyOptions` feature gate.

The policy options are split into two groups: alpha quality (hidden by default) and beta quality (visible by default). The groups are guarded respectively by the `CPUManagerPolicyAlphaOptions` and `CPUManagerPolicyBetaOptions` feature gates. Diverging from the Kubernetes standard, these feature gates guard groups of options, because it would have been too cumbersome to add a feature gate for each individual option.

Changing the CPU Manager Policy

Since the CPU manger policy can only be applied when kubelet spawns new pods, simply changing from "none" to "static" won't apply to existing pods. So in order to properly change the CPU manager policy on a node, perform the following steps:

1. Drain the node.
2. Stop kubelet.
3. Remove the old CPU manager state file. The path to this file is `/var/lib/kubelet/cpu_manager_state` by default. This clears the state

maintained by the CPUManager so that the cpu-sets set up by the new policy won't conflict with it.
4. Edit the kubelet configuration to change the CPU manager policy to the desired value.
5. Start kubelet.

Repeat this process for every node that needs its CPU manager policy changed. Skipping this process will result in kubelet crashlooping with the following error:

```
could not restore state from checkpoint: configured policy
"static" differs from state checkpoint policy "none", please
drain this node and delete the CPU manager checkpoint file "/var/
lib/kubelet/cpu_manager_state" before restarting Kubelet
```

None policy

The `none` policy explicitly enables the existing default CPU affinity scheme, providing no affinity beyond what the OS scheduler does automatically. Â Limits on CPU usage for Guaranteed pods and Burstable 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. Â The exclusivity only extends to other pods.
Note: CPU Manager doesn't support offlining and onlining of CPUs at runtime. Also, if the set of online CPUs changes on the node, the node must be drained and CPU manager manually reset by deleting the state file `cpu_manager_state` in the kubelet root directory.

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. From 1.17, the CPU reservation list can be specified explicitly by kubelet `--reserved-cpus` option. The explicit CPU list specified by `--reserved-cpus` takes precedence over the CPU reservation specified by `--kube-reserved` and `--system-reserved`. CPUs reserved by these options are taken, in integer quantity, from the initial shared pool in ascending order by physical core ID. Â 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` or `--reserved-cpus` 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.Â 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.

```yaml
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.

```yaml
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.

```yaml
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.

Static policy options

You can toggle groups of options on and off based upon their maturity level using the following feature gates:

- `CPUManagerPolicyBetaOptions` default enabled. Disable to hide beta-level options.
- `CPUManagerPolicyAlphaOptions` default disabled. Enable to show alpha-level options. You will still have to enable each option using the `CPUManagerPolicyOptions` kubelet option.

The following policy options exist for the static `CPUManager` policy:

- `full-pcpus-only` (beta, visible by default)
- `distribute-cpus-across-numa` (alpha, hidden by default)

If the `full-pcpus-only` policy option is specified, the static policy will always allocate full physical cores. By default, without this option, the static policy allocates CPUs using a topology-aware best-fit allocation. On SMT enabled systems, the policy can allocate individual virtual cores, which correspond to hardware threads. This can lead to different containers sharing the same physical cores; this behaviour in turn contributes to the noisy neighbours problem. With the option enabled, the pod will be admitted by the kubelet only if the CPU request of all its containers can be fulfilled by allocating full physical cores. If the pod does not pass the admission, it will be put in Failed state with the message `SMTAlignmentError`.

If the `distribute-cpus-across-numa`policy option is specified, the static policy will evenly distribute CPUs across NUMA nodes in cases where more than one NUMA node is required to satisfy the allocation. By default, the `CPUManager` will pack CPUs onto one NUMA node until it is filled, with any remaining CPUs simply spilling over to the next NUMA node. This can cause undesired bottlenecks in parallel code relying on barriers (and similar synchronization primitives), as this type of code tends to run only as fast as its slowest worker (which is slowed down by the fact that fewer CPUs are available on at least one NUMA node). By distributing CPUs evenly across NUMA nodes, application developers can more easily ensure that no single worker suffers from NUMA effects more than any other, improving the overall performance of these types of applications.

The `full-pcpus-only` option can be enabled by adding `full-pcups-only=true` to the CPUManager policy options. Likewise, the `distribute-cpus-across-numa` option can be enabled by adding `distribute-cpus-across-numa=true` to the CPUManager policy options. When both are set, they are "additive" in the sense that CPUs will be distributed across NUMA nodes in chunks of full-pcpus rather than individual cores.

Control Topology Management Policies on a node

FEATURE STATE: `Kubernetes v1.18 [beta]`

An increasing number of systems leverage a combination of CPUs and hardware accelerators to support latency-critical execution and high-throughput parallel computation. These include workloads in fields such as telecommunications, scientific computing, machine learning, financial services and data analytics. Such hybrid systems comprise a high performance environment.

In order to extract the best performance, optimizations related to CPU isolation, memory and device locality are required. However, in Kubernetes, these optimizations are handled by a disjoint set of components.

Topology Manager is a Kubelet component that aims to co-ordinate the set of components that are responsible for these optimizations.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.18. To check the version, enter `kubectl version`.

How Topology Manager Works

Prior to the introduction of Topology Manager, the CPU and Device Manager in Kubernetes make resource allocation decisions independently of each other. This can result in undesirable allocations on multiple-socketed systems, performance/latency sensitive applications will suffer due to these undesirable allocations. Undesirable in this case meaning for example, CPUs and devices being allocated from different NUMA Nodes thus, incurring additional latency.

The Topology Manager is a Kubelet component, which acts as a source of truth so that other Kubelet components can make topology aligned resource allocation choices.

The Topology Manager provides an interface for components, called *Hint Providers*, to send and receive topology information. Topology Manager has a set of node level policies which are explained below.

The Topology manager receives Topology information from the *Hint Providers* as a bitmask denoting NUMA Nodes available and a preferred allocation indication. The Topology Manager policies perform a set of operations on the hints provided and converge on the hint determined by the policy to give the optimal result, if an undesirable hint is stored the preferred field for the hint will be set to false. In the current policies preferred is the narrowest preferred mask. The selected hint is stored as part of the Topology Manager. Depending on the policy configured the pod can be accepted or rejected from the node based on the selected hint. The hint is then stored in the Topology Manager for use by the *Hint Providers* when making the resource allocation decisions.

Enable the Topology Manager feature

Support for the Topology Manager requires `TopologyManager` feature gate to be enabled. It is enabled by default starting with Kubernetes 1.18.

Topology Manager Scopes and Policies

The Topology Manager currently:

- Aligns Pods of all QoS classes.
- Aligns the requested resources that Hint Provider provides topology hints for.

If these conditions are met, the Topology Manager will align the requested resources.

In order to customise how this alignment is carried out, the Topology Manager provides two distinct knobs: `scope` and `policy`.

The `scope` defines the granularity at which you would like resource alignment to be performed (e.g. at the `pod` or `container` level). And the `policy` defines the actual strategy used to carry out the alignment (e.g. `best-effort`, `restricted`, `single-numa-node`, etc.).

Details on the various `scopes` and `policies` available today can be found below.

Note: To align CPU resources with other requested resources in a Pod Spec, the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See control CPU Management Policies.
Note: To align memory (and hugepages) resources with other requested resources in a Pod Spec, the Memory Manager should be enabled and proper Memory Manager policy should be configured on a Node. Examine Memory Manager documentation.

Topology Manager Scopes

The Topology Manager can deal with the alignment of resources in a couple of distinct scopes:

- `container` (default)
- `pod`

Either option can be selected at a time of the kubelet startup, with `--topology-manager-scope` flag.

container scope

The `container` scope is used by default.

Within this scope, the Topology Manager performs a number of sequential resource alignments, i.e., for each container (in a pod) a separate alignment is computed. In other words, there is no notion of grouping the containers to a specific set of NUMA nodes, for this particular scope. In effect, the Topology Manager performs an arbitrary alignment of individual containers to NUMA nodes.

The notion of grouping the containers was endorsed and implemented on purpose in the following scope, for example the `pod` scope.

pod scope

To select the `pod` scope, start the kubelet with the command line option `--topology-manager-scope=pod`.

This scope allows for grouping all containers in a pod to a common set of NUMA nodes. That is, the Topology Manager treats a pod as a whole and attempts to allocate the entire pod (all containers) to either a single NUMA node or a common set of NUMA nodes. The following examples illustrate the alignments produced by the Topology Manager on different occasions:

- all containers can be and are allocated to a single NUMA node;
- all containers can be and are allocated to a shared set of NUMA nodes.

The total amount of particular resource demanded for the entire pod is calculated according to effective requests/limits formula, and thus, this total value is equal to the maximum of:

- the sum of all app container requests,
- the maximum of init container requests, for a resource.

Using the `pod` scope in tandem with `single-numa-node` Topology Manager policy is specifically valuable for workloads that are latency sensitive or for high-throughput applications that perform IPC. By combining both options, you are able to place all containers in a pod onto a single NUMA node; hence, the inter-NUMA communication overhead can be eliminated for that pod.

In the case of `single-numa-node` policy, a pod is accepted only if a suitable set of NUMA nodes is present among possible allocations. Reconsider the example above:

- a set containing only a single NUMA node - it leads to pod being admitted,
- whereas a set containing more NUMA nodes - it results in pod rejection (because instead of one NUMA node, two or more NUMA nodes are required to satisfy the allocation).

To recap, Topology Manager first computes a set of NUMA nodes and then tests it against Topology Manager policy, which either leads to the rejection or admission of the pod.

Topology Manager Policies

Topology Manager supports four allocation policies. You can set a policy via a Kubelet flag, `--topology-manager-policy`. There are four supported policies:

- `none` (default)
- `best-effort`
- `restricted`
- `single-numa-node`

Note: If Topology Manager is configured with the **pod** scope, the container, which is considered by the policy, is reflecting requirements of the entire pod, and thus each container from the pod will result with **the same** topology alignment decision.

none policy

This is the default policy and does not perform any topology alignment.

best-effort policy

For each container in a Pod, the kubelet, with `best-effort` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will store this and admit the pod to the node anyway.

The *Hint Providers* can then use this information when making the resource allocation decision.

restricted policy

For each container in a Pod, the kubelet, with `restricted` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will reject this pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a ReplicaSet or Deployment to trigger a redeploy of the pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

If the pod is admitted, the *Hint Providers* can then use this information when making the resource allocation decision.

single-numa-node policy

For each container in a Pod, the kubelet, with `single-numa-node` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager determines if a single NUMA Node affinity is possible. If it is, Topology Manager will store this and the *Hint Providers* can then use this information when making the resource allocation decision. If, however, this is not possible then the Topology Manager will reject the pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a Deployment with replicas to trigger a redeploy of the Pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

Pod Interactions with Topology Manager Policies

Consider the containers in the following pod specs:

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
```

This pod runs in the `BestEffort` QoS class because no resource `requests` or `limits` are specified.

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

This pod runs in the `Burstable` QoS class because requests are less than limits.

If the selected policy is anything other than `none`, Topology Manager would consider these Pod specifications. The Topology Manager would consult the Hint Providers to get topology hints. In the case of the `static`, the CPU Manager policy would return default topology hint, because these Pods do not have explicitly request CPU resources.

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

This pod with integer CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
```

This pod with sharing CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        example.com/deviceA: "1"
        example.com/deviceB: "1"
      requests:
        example.com/deviceA: "1"
        example.com/deviceB: "1"
```

This pod runs in the `BestEffort` QoS class because there are no CPU and memory requests.

The Topology Manager would consider the above pods. The Topology Manager would consult the Hint Providers, which are CPU and Device Manager to get topology hints for the pods.

In the case of the `Guaranteed` pod with integer CPU request, the `static` CPU Manager policy would return topology hints relating to the exclusive CPU and the Device Manager would send back hints for the requested device.

In the case of the `Guaranteed` pod with sharing CPU request, the `static` CPU Manager policy would return default topology hint as there is no exclusive CPU request and the Device Manager would send back hints for the requested device.

In the above two cases of the `Guaranteed` pod, the `none` CPU Manager policy would return default topology hint.

In the case of the `BestEffort` pod, the `static` CPU Manager policy would send back the default topology hint as there is no CPU request and the Device Manager would send back the hints for each of the requested devices.

Using this information the Topology Manager calculates the optimal hint for the pod and stores this information, which will be used by the Hint Providers when they are making their resource assignments.

Known Limitations

1. The maximum number of NUMA nodes that Topology Manager allows is 8. With more than 8 NUMA nodes there will be a state explosion when trying to enumerate the possible NUMA affinities and generating their hints.

2. The scheduler is not topology-aware, so it is possible to be scheduled on a node and then fail on the node due to the Topology Manager.

Customizing DNS Service

This page explains how to configure your DNS Pod(s) and customize the DNS resolution process in your 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. It is recommended to

run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your cluster must be running the CoreDNS add-on. Migrating to CoreDNS explains how to use `kubeadm` to migrate from `kube-dns`.

Your Kubernetes server must be at or later than version v1.12. To check the version, enter `kubectl version`.

Introduction

DNS is a built-in Kubernetes service launched automatically using the *addon manager* cluster add-on.

As of Kubernetes v1.12, CoreDNS is the recommended DNS Server, replacing kube-dns. If your cluster originally used kube-dns, you may still have `kube-dns` deployed rather than CoreDNS.

Note: The CoreDNS Service is named `kube-dns` in the `metadata.name` field. This is so that there is greater interoperability with workloads that relied on the legacy `kube-dns` Service name to resolve addresses internal to the cluster. Using a Service named `kube-dns` abstracts away the implementation detail of which DNS provider is running behind that common name.

If you are running CoreDNS as a Deployment, it will typically be exposed as a Kubernetes Service with a static IP address. The kubelet passes DNS resolver information to each container with the `--cluster-dns=<dns-service-ip>` flag.

DNS names also need domains. You configure the local domain in the kubelet with the flag `--cluster-domain=<default-local-domain>`.

The DNS server supports forward lookups (A and AAAA records), port lookups (SRV records), reverse IP address lookups (PTR records), and more. For more information, see DNS for Services and Pods.

If a Pod's `dnsPolicy` is set to `default`, it inherits the name resolution configuration from the node that the Pod runs on. The Pod's DNS resolution should behave the same as the node. But see Known issues.

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. Set this flag to "" to prevent Pods from inheriting DNS. Set it to a valid file path to specify a file other than `/etc/resolv.conf` for DNS inheritance.

CoreDNS

CoreDNS is a general-purpose authoritative DNS server that can serve as cluster DNS, complying with the dns specifications.

CoreDNS ConfigMap options

CoreDNS is a DNS server that is modular and pluggable, and each plugin adds new functionality to CoreDNS. This can be configured by maintaining a Corefile, which is the CoreDNS configuration file. As a cluster administrator, you can modify the ConfigMap for the CoreDNS Corefile to change how DNS service discovery behaves for that cluster.

In Kubernetes, CoreDNS is installed with the following default Corefile configuration:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health {
            lameduck 5s
        }
        ready
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
            ttl 30
        }
        prometheus :9153
        forward . /etc/resolv.conf
        cache 30
        loop
        reload
        loadbalance
    }
```

The Corefile configuration includes the following plugins of CoreDNS:

- errors: Errors are logged to stdout.
- health: Health of CoreDNS is reported to `http://localhost:8080/health`. In this extended syntax `lameduck` will make the process unhealthy then wait for 5 seconds before the process is shut down.
- ready: An HTTP endpoint on port 8181 will return 200 OK, when all plugins that are able to signal readiness have done so.
- kubernetes: CoreDNS will reply to DNS queries based on IP of the services and pods of Kubernetes. You can find more details about that

plugin on the CoreDNS website. `ttl` allows you to set a custom TTL for responses. The default is 5 seconds. The minimum TTL allowed is 0 seconds, and the maximum is capped at 3600 seconds. Setting TTL to 0 will prevent records from being cached.
The `pods insecure` option is provided for backward compatibility with *kube-dns*. You can use the `pods verified` option, which returns an A record only if there exists a pod in same namespace with matching IP. The `pods disabled` option can be used if you don't use pod records.
- prometheus: Metrics of CoreDNS are available at `http://localhost:9153/metrics` in Prometheus format (also known as OpenMetrics).
- forward: Any queries that are not within the cluster domain of Kubernetes will be forwarded to predefined resolvers (/etc/resolv.conf).
- cache: This enables a frontend cache.
- loop: Detects simple forwarding loops and halts the CoreDNS process if a loop is found.
- reload: Allows automatic reload of a changed Corefile. After you edit the ConfigMap configuration, allow two minutes for your changes to take effect.
- loadbalance: This is a round-robin DNS loadbalancer that randomizes the order of A, AAAA, and MX records in the answer.

You can modify the default CoreDNS behavior by modifying the ConfigMap.

Configuration of Stub-domain and upstream nameserver using CoreDNS

CoreDNS has the ability to configure stubdomains and upstream nameservers using the forward plugin.

Example

If a cluster operator has a Consul domain server located at 10.150.0.1, and all Consul names have the suffix .consul.local. To configure it in CoreDNS, the cluster administrator creates the following stanza in the CoreDNS ConfigMap.

```
consul.local:53 {
        errors
        cache 30
        forward . 10.150.0.1
    }
```

To explicitly force all non-cluster DNS lookups to go through a specific nameserver at 172.16.0.1, point the `forward` to the nameserver instead of `/etc/resolv.conf`

```
forward .  172.16.0.1
```

The final ConfigMap along with the default `Corefile` configuration looks like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
           pods insecure
           fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . 172.16.0.1
        cache 30
        loop
        reload
        loadbalance
    }
    consul.local:53 {
        errors
        cache 30
        forward . 10.150.0.1
    }
```

The `kubeadm` tool supports automatic translation from the kube-dns ConfigMap to the equivalent CoreDNS ConfigMap.

Note: While kube-dns accepts an FQDN for stubdomain and nameserver (eg: ns.foo.com), CoreDNS does not support this feature. During translation, all FQDN nameservers will be omitted from the CoreDNS config.

CoreDNS configuration equivalent to kube-dns

CoreDNS supports the features of kube-dns and more. A ConfigMap created for kube-dns to support `StubDomains`and `upstreamNameservers` translates to the `forward` plugin in CoreDNS.

Example

This example ConfigMap for kube-dns specifies stubdomains and upstreamnameservers:

```
apiVersion: v1
data:
  stubDomains: |
        {"abc.com" : ["1.2.3.4"], "my.cluster.local" :
["2.3.4.5"]}
```

```
  upstreamNameservers: |
        ["8.8.8.8", "8.8.4.4"]
kind: ConfigMap
```

The equivalent configuration in CoreDNS creates a Corefile:

- For stubDomains:

```
abc.com:53 {
    errors
    cache 30
    forward . 1.2.3.4
}
my.cluster.local:53 {
    errors
    cache 30
    forward . 2.3.4.5
}
```

The complete Corefile with the default plugins:

```
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    federation cluster.local {
        foo foo.feddomain.com
    }
    prometheus :9153
    forward . 8.8.8.8 8.8.4.4
    cache 30
}
abc.com:53 {
    errors
    cache 30
    forward . 1.2.3.4
}
my.cluster.local:53 {
    errors
    cache 30
    forward . 2.3.4.5
}
```

Migration to CoreDNS

To migrate from kube-dns to CoreDNS, a detailed blog article is available to help users adapt CoreDNS in place of kube-dns.

You can also migrate using the official CoreDNS deploy script.

What's next

- Read Debugging DNS Resolution

Debugging DNS Resolution

This page provides hints on diagnosing DNS problems.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your cluster must be configured to use the CoreDNS addon or its precursor, kube-dns.

Your Kubernetes server must be at or later than version v1.6. To check the version, enter `kubectl version`.

Create a simple Pod to use as a test environment

admin/dns/dnsutils.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dnsutils
  namespace: default
spec:
  containers:
  - name: dnsutils
    image: k8s.gcr.io/e2e-test-images/jessie-dnsutils:1.3
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

Note: This example creates a pod in the `default` namespace. DNS name resolution for services depends on the namespace of the pod. For more information, review DNS for Services and Pods.

Use that manifest to create a Pod:

```
kubectl apply -f https://k8s.io/examples/admin/dns/dnsutils.yaml
```

```
pod/dnsutils created
```

â€¦and verify its status:

```
kubectl get pods dnsutils
```

```
NAME      READY     STATUS    RESTARTS   AGE
dnsutils   1/1       Running   0          <some-time>
```

Once that Pod is running, you can exec `nslookup` in that environment. If you see something like the following, DNS is working correctly.

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes.default
Address 1: 10.0.0.1
```

If the `nslookup` command fails, check the following:

Check the local DNS configuration first

Take a look inside the resolv.conf file. (See Customizing DNS Service and Known issues below for more information)

```
kubectl exec -ti dnsutils -- 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.gce_project_id.internal
nameserver 10.0.0.10
options ndots:5
```

Errors such as the following indicate a problem with the CoreDNS (or kube-dns) add-on or with associated Services:

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10

nslookup: can't resolve 'kubernetes.default'
```

or

```
kubectl exec -i -t dnsutils -- 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
```

```
NAME                       READY     STATUS    RESTARTS   AGE
...
coredns-7b96bf9f76-5hsxb   1/1       Running   0           1h
coredns-7b96bf9f76-mvmmt   1/1       Running   0           1h
...
```

Note: The value for label `k8s-app` is `kube-dns` for both CoreDNS and kube-dns deployments.

If you see that no CoreDNS 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 the `kubectl logs` command to see logs for the DNS containers.

For CoreDNS:

```
kubectl logs --namespace=kube-system -l k8s-app=kube-dns
```

Here is an example of a healthy CoreDNS log:

```
.:53
2018/08/15 14:37:17 [INFO] CoreDNS-1.2.2
2018/08/15 14:37:17 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.2
linux/amd64, go1.10.3, 2e322f6
2018/08/15 14:37:17 [INFO] plugin/reload: Running configuration
MD5 = 24e6c59e83ce706f07bcc82c31b1ea1c
```

See if there are any suspicious or unexpected messages in the logs.

Is DNS service up?

Verify that the DNS service is up by using the `kubectl get service` command.

```
kubectl get svc --namespace=kube-system
```

```
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)             AGE
```

```
...
kube-dns     ClusterIP   10.0.0.10      <none>        53/UDP,53/
TCP        1h
...
```

Note: The service name is `kube-dns` for both CoreDNS and kube-dns deployments.

If you have created the Service or in the case it should be created by default but it does not appear, see debugging Services for more information.

Are DNS endpoints exposed?

You can verify that DNS endpoints are exposed by using the `kubectl get endpoints` command.

```
kubectl get endpoints kube-dns --namespace=kube-system
```

```
NAME       ENDPOINTS                       AGE
kube-dns   10.180.3.17:53,10.180.3.17:53    1h
```

If you do not see the endpoints, see the endpoints section in the debugging Services documentation.

For additional Kubernetes DNS examples, see the cluster-dns examples in the Kubernetes GitHub repository.

Are DNS queries being received/processed?

You can verify if queries are being received by CoreDNS by adding the `log` plugin to the CoreDNS configuration (aka Corefile). The CoreDNS Corefile is held in a ConfigMap named `coredns`. To edit it, use the command:

```
kubectl -n kube-system edit configmap coredns
```

Then add `log` in the Corefile section per the example below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        log
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          upstream
          fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf
        cache 30
        loop
        reload
        loadbalance
    }
```

After saving the changes, it may take up to minute or two for Kubernetes to propagate these changes to the CoreDNS pods.

Next, make some queries and view the logs per the sections above in this document. If CoreDNS pods are receiving the queries, you should see them in the logs.

Here is an example of a query in the log:

```
.:53
2018/08/15 14:37:15 [INFO] CoreDNS-1.2.0
2018/08/15 14:37:15 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.0
linux/amd64, go1.10.3, 2e322f6
2018/09/07 15:29:04 [INFO] plugin/reload: Running configuration
MD5 = 162475cdf272d8aa601e6fe67a6ad42f
2018/09/07 15:29:04 [INFO] Reloading complete
172.17.0.18:41675 - [07/Sep/2018:15:29:11 +0000] 59925 "A IN
kubernetes.default.svc.cluster.local. udp 54 false 512" NOERROR
qr,aa,rd,ra 106 0.000066649s
```

Are you in the right namespace for the service?

DNS queries that don't specify a namespace are limited to the pod's namespace.

If the namespace of the pod and service differ, the DNS query must include the namespace of the service.

This query is limited to the pod's namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-name>
```

This query specifies the namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-
name>.<namespace>
```

To learn more about name resolution, see DNS for Services and Pods.

Known issues

Some Linux distributions (e.g. Ubuntu) use a local DNS resolver by default (systemd-resolved). Systemd-resolved moves and replaces `/etc/`

`resolv.conf` with a stub file that can cause a fatal forwarding loop when resolving names in upstream servers. This can be fixed manually by using kubelet's `--resolv-conf` flag to point to the correct `resolv.conf` (With `systemd-resolved`, this is `/run/systemd/resolve/resolv.conf`). kubeadm automatically detects `systemd-resolved`, and adjusts the kubelet flags accordingly.

Kubernetes installs do not configure the nodes' `resolv.conf` files to use the cluster DNS by default, because that process is inherently distribution-specific. This should probably be implemented eventually.

Linux's libc (a.k.a. glibc) has a limit for the DNS `nameserver` records to 3 by default. What's more, for the glibc versions which are older than glibc-2.17-222 (the new versions update see this issue), the allowed number of DNS `search` records has been limited to 6 (see this bug from 2005). 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 while your glibc version is in the affected list, some of those settings will be lost. To work around the DNS `nameserver` records limit, the node can run `dnsmasq`, which will provide more `nameserver` entries. You can also use kubelet's `--resolv-conf` flag. To fix the DNS `search` records limit, consider upgrading your linux distribution or upgrading to an unaffected version of glibc.

Note: With Expanded DNS Configuration, Kubernetes allows more DNS `search` records.

If you are using Alpine version 3.3 or earlier as your base image, DNS may not work properly due to a known issue with Alpine. Kubernetes issue 30215 details more information on this.

What's next

- See Autoscaling the DNS Service in a Cluster.
- Read DNS for Services and Pods

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.

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.8. To check the version, enter `kubectl version`.

Make sure you've configured a network provider with network policy support. There are a number of network providers that support NetworkPolicy, including:

- Antrea
- Calico
- Cilium
- Kube-router
- Romana
- Weave Net

Create an `nginx` deployment and expose it via a service

To see how Kubernetes network policy works, start off by creating an `nginx` Deployment.

```
kubectl create deployment nginx --image=nginx
```

```
deployment.apps/nginx created
```

Expose the Deployment through a Service called `nginx`.

```
kubectl expose deployment nginx --port=80
```

```
service/nginx exposed
```

The above commands create a Deployment with an nginx Pod and expose the Deployment through a Service named `nginx`. The `nginx` Pod and Deployment are found in the `default` namespace.

```
kubectl get svc,pod
```

```
NAME                        CLUSTER-IP    EXTERNAL-IP
PORT(S)    AGE
service/kubernetes          10.100.0.1    <none>        443/
TCP    46m
service/nginx               10.100.0.16   <none>        80/
TCP     33s
```

```
NAME                        READY         STATUS
RESTARTS   AGE
pod/nginx-701339712-e0qfq   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 access the `nginx` Service from another Pod in the `default` namespace, start a busybox container:

```
kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh
```

In your shell, run the following command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

Limit access to the `nginx` service

To limit the access to the `nginx` service so that only Pods with the label `access: true` can query it, create a NetworkPolicy object as follows:

service/networking/nginx-policy.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

The name of a NetworkPolicy object must be a valid DNS subdomain name.

Note: NetworkPolicy includes a `podSelector` which selects the grouping of Pods to which the policy applies. You can see this policy selects Pods with the label `app=nginx`. The label was automatically added to the Pod in the `nginx` Deployment. An empty `podSelector` selects all pods in the namespace.

Assign the policy to the service

Use kubectl to create a NetworkPolicy from the above `nginx-policy.yaml` file:

```
kubectl apply -f https://k8s.io/examples/service/networking/
nginx-policy.yaml
```

```
networkpolicy.networking.k8s.io/access-nginx created
```

Test access to the service when access label is not defined

When you attempt to access the `nginx` Service from a Pod without the correct labels, the request times out:

```
kubectl run busybox --rm -ti --image=busybox:1.28 -- /bin/sh
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
wget: download timed out
```

Define access label and test again

You can create a Pod with the correct labels to see that the request is allowed:

```
kubectl run busybox --rm -ti --labels="access=true" --
image=busybox:1.28 -- /bin/sh
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

Developing Cloud Controller Manager

FEATURE STATE: `Kubernetes v1.11 [beta]`

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

Background

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 Kubernetes project provides skeleton cloud-controller-manager code with Go interfaces to allow you (or your cloud provider) to plug in your own implementations. This means that a cloud provider can implement a cloud-controller-manager by importing packages from Kubernetes core; each cloudprovider will register their own code by calling `cloudprovider.RegisterCloudProvider` to update a global variable of available cloud providers.

Developing

Out of tree

To build an out-of-tree cloud-controller-manager for your cloud:

1. Create a go package with an implementation that satisfies cloudprovider.Interface.
2. Use `main.go` in cloud-controller-manager from Kubernetes 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`.

Many cloud providers publish their controller manager code as open source. If you are creating a new cloud-controller-manager from scratch, you could take an existing out-of-tree cloud controller manager as your starting point.

In tree

For in-tree cloud providers, you can run the in-tree cloud controller manager as a DaemonSet in your cluster. See Cloud Controller Manager Administration for more details.

Enable Or Disable A Kubernetes API

This page shows how to enable or disable an API version from your cluster's control plane.

Specific API versions can be turned on or off by passing `--runtime-config=api/<version>` as a command line argument to the API server. The values for this argument are a comma-separated list of API versions. Later values override earlier values.

The `runtime-config` command line argument also supports 2 special keys:

- `api/all`, representing all known APIs
- `api/legacy`, representing only legacy APIs. Legacy APIs are any APIs that have been explicitly deprecated.

For example, to turning off all API versions except v1, pass `--runtime-config=api/all=false,api/v1=true` to the `kube-apiserver`.

What's next

Read the full documentation for the `kube-apiserver` component.

Enabling Service Topology

FEATURE STATE: `Kubernetes v1.21 [deprecated]`

This feature, specifically the alpha `topologyKeys` field, is deprecated since Kubernetes v1.21. Topology Aware Hints, introduced in Kubernetes v1.21, provide similar functionality.

Service Topology enables a Service to route traffic based upon the Node topology of the cluster. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

The following prerequisites are needed in order to enable topology aware service routing:

- Kubernetes v1.17 or later
- Configure kube-proxy to run in iptables mode or IPVS mode

Enable Service Topology

FEATURE STATE: `Kubernetes v1.21 [deprecated]`

To enable service topology, enable the `ServiceTopology` feature gate for all Kubernetes components:

```
--feature-gates="ServiceTopology=true`
```

What's next

- Read about Topology Aware Hints, the replacement for the `topologyKeys` field.
- Read about EndpointSlices
- Read about the Service Topology concept
- Read Connecting Applications with Services

Encrypting Secret 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

  Your Kubernetes server must be at or later than version 1.13. To check the version, enter `kubectl version`.

- etcd v3.0 or later is required

Configuration and determining whether encryption at rest is already enabled

The `kube-apiserver` process accepts an argument `--encryption-provider-config` that controls how API data is encrypted in etcd. The configuration is provided as an API named [`EncryptionConfiguration`](). An example configuration is provided below.

Caution: IMPORTANT: For high-availability configurations (with two or more control plane nodes), the encryption configuration file must be the same! Otherwise, the `kube-apiserver` component cannot decrypt data stored in the etcd.

Understanding the encryption at rest configuration.

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
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 written into the storage. When reading resources from storage, each provider that matches the stored data attempts in order to decrypt the data. 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.

For more detailed information about the `EncryptionConfiguration` struct, please refer to the encryption configuration API.

Caution: 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.
`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.
`aescbc`	AES-CBC with PKCS#7 padding	Weak	Fast	32-byte	Not recommended due to CBC's vulnerability to padding oracle attacks.

Name	Encryption	Strength	Speed	Key Length	Other Considerations
`kms`	Uses envelope encryption scheme: Data is encrypted by data encryption keys (DEKs) using AES-CBC with PKCS#7 padding, DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS)	Strongest	Fast	32-bytes	The recommended choice for using a third party tool for key management. Simplifies key rotation, with a new DEK generated for each encryption, and KEK rotation controlled by the user. Configure the KMS provider

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.

Caution: Storing the raw encryption key in the EncryptionConfig only moderately improves your security posture, compared to no encryption. Please use `kms` provider for additional security.

By default, the `identity` provider is used to protect Secrets in etcd, which provides no encryption. `EncryptionConfiguration` was introduced to encrypt Secrets locally, with a locally managed key.

Encrypting Secrets with a locally managed key protects against an etcd compromise, but it fails to protect against a host compromise. Since the encryption keys are stored on the host in the EncryptionConfiguration YAML file, a skilled attacker can access that file and extract the encryption keys.

Envelope encryption creates dependence on a separate key, not stored in Kubernetes. In this case, an attacker would need to compromise etcd, the `kubeapi-server`, and the third-party KMS provider to retrieve the plaintext values, providing a higher level of security than locally stored encryption keys.

Encrypting your data

Create a new encryption config file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
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 macOS, run the following command:

   ```
   head -c 32 /dev/urandom | base64
   ```

2. Place that value in the `secret` field of the `EncryptionConfiguration` struct.

3. Set the `--encryption-provider-config` flag on the `kube-apiserver` to point to the location of the config file.

4. Restart your API server.

Caution: Your config file contains keys that can decrypt the contents in etcd, so you must properly restrict permissions on your control-plane nodes 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 this, 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` command line, read that Secret out of etcd:

   ```
   ETCDCTL_API=3 etcdctl get /registry/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
   ```

The output should contain `mykey: bXlkYXRh`, with contents of `mydata` encoded, check [decoding a Secret](#) to completely decode the Secret.

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.

Note: 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 a 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 --all-namespaces -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 have backed up etcd with the new key in use and updated all Secrets

When running a single `kube-apiserver` instance, step 2 may be skipped.

Decrypting all data

To disable encryption at rest, place the `identity` provider as the first entry in the config and restart all `kube-apiserver` processes.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - identity: {}
      - aescbc:
          keys:
            - name: key1
              secret: <BASE 64 ENCODED SECRET>
```

Then run the following command to force decrypt all Secrets:

```
kubectl get secrets --all-namespaces -o json | kubectl replace -
f -
```

What's next

- Learn more about the EncryptionConfiguration configuration API (v1).

Guaranteed Scheduling For Critical Add-On Pods

Kubernetes core components such as the API server, scheduler, and controller-manager run on a control plane node. However, add-ons must run on a regular cluster node. Some of these add-ons are critical to a fully functional cluster, such as metrics-server, 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).

Note that marking a pod as critical is not meant to prevent evictions entirely; it only prevents the pod from becoming permanently unavailable. A static pod marked as critical, can't be evicted. However, a non-static pods marked as critical are always rescheduled.

Marking pod as critical

To mark a Pod as critical, set priorityClassName for that Pod to `system-cluster-critical` or `system-node-critical`. `system-node-critical` is the highest available priority, even higher than `system-cluster-critical`.

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. It is recommended to

run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

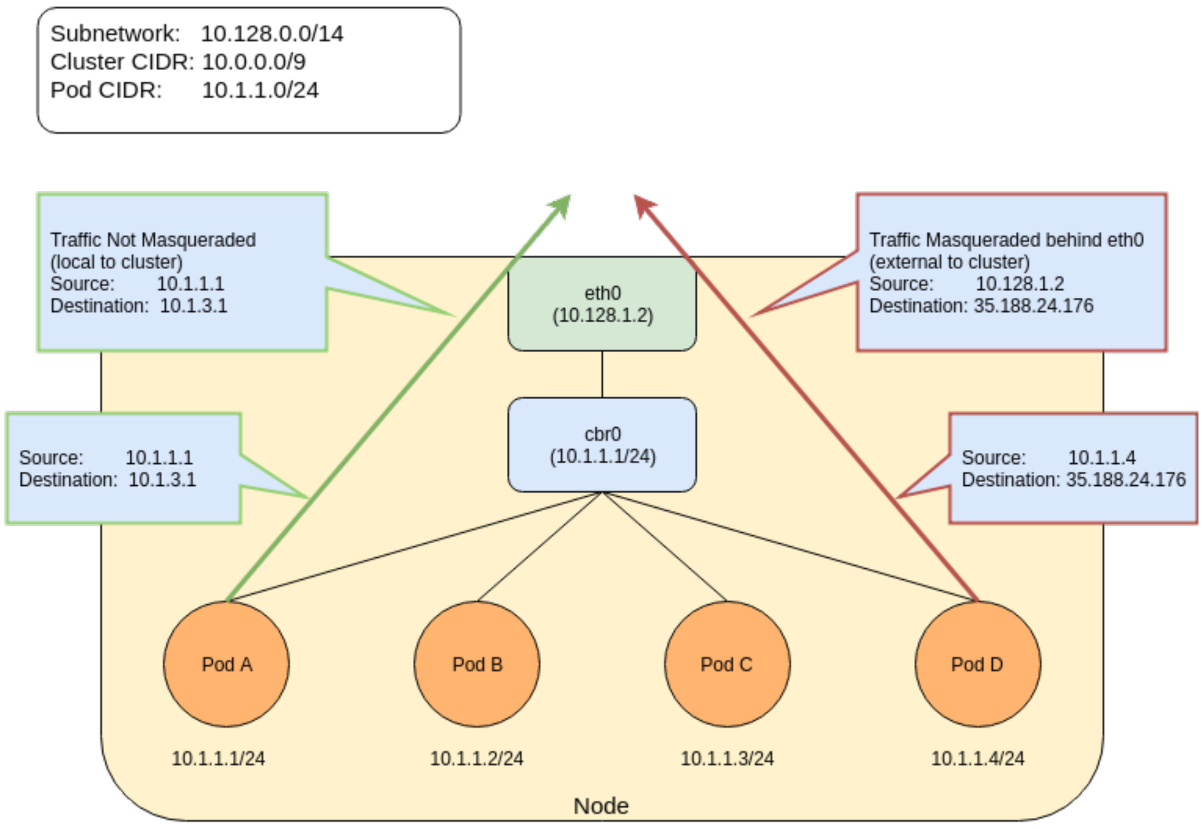
IP Masquerade Agent User Guide

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 Google Kubernetes Engine, 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) Source: 10.1.1.1 Destination: 10.1.3.1
Traffic Masqueraded behind eth0 (external to cluster) Source: 10.128.1.2 Destination: 35.188.24.176
eth0 (10.128.1.2)
cbr0 (10.1.1.1/24)
Source: 10.1.1.1 Destination: 10.1.3.1
Source: 10.1.1.4 Destination: 35.188.24.176
Pod A
Pod B
Pod C
Pod D
10.1.1.1/24
10.1.1.2/24
10.1.1.3/24
10.1.1.4/24
Node

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`: A time interval at which the agent attempts to reload config from disk. For example: '30s', where 's' means seconds, 'ms' means milliseconds.

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: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             10.0.0.0/8           /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
```

```
RETURN     all  --  anywhere             172.16.0.0/12       /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             192.168.0.0/16      /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
MASQUERADE  all  --  anywhere             anywhere             /
* ip-masq-agent: outbound traffic should be subject to
MASQUERADE (this match must come after cluster-local CIDR
matches) */ ADDRTYPE match dst-type !LOCAL
```

By default, in GCE/Google Kubernetes Engine, 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.

Create an ip-masq-agent

To create an ip-masq-agent, run the following kubectl command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-
sigs/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 node.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 `resyncInterval` 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: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN     all  --  anywhere             10.0.0.0/8           /*
ip-masq-agent: cluster-local
MASQUERADE  all  --  anywhere             anywhere             /
* ip-masq-agent: outbound traffic should be subject to
MASQUERADE (this match must come after cluster-local CIDR
matches) */ ADDRTYPE match dst-type !LOCAL
```

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 ConfigMap.

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
masqLinkLocal: true
```

Limit Storage Consumption

This example demonstrates how 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

```yaml
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.

Migrate Replicated Control Plane To Use Cloud Controller Manager

FEATURE STATE: `Kubernetes v1.22 [beta]`

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

Background

As part of the cloud provider extraction effort, all cloud specific controllers must be moved out of the `kube-controller-manager`. All existing clusters that run cloud controllers in the `kube-controller-manager` must migrate to instead run the controllers in a cloud provider specific `cloud-controller-manager`.

Leader Migration provides a mechanism in which HA clusters can safely migrate "cloud specific" controllers between the `kube-controller-manager` and the `cloud-controller-manager` via a shared resource lock between the two components while upgrading the replicated control plane. For a single-node control plane, or if unavailability of controller managers can be tolerated during the upgrade, Leader Migration is not needed and this guide can be ignored.

Leader Migration can be enabled by setting `--enable-leader-migration` on `kube-controller-manager` or `cloud-controller-manager`. Leader Migration only applies during the upgrade and can be safely disabled or left enabled after the upgrade is complete.

This guide walks you through the manual process of upgrading the control plane from `kube-controller-manager` with built-in cloud provider to running both `kube-controller-manager` and `cloud-controller-manager`. If you use a tool to administrator the cluster, please refer to the documentation of the tool and the cloud provider for more details.

Before you begin

It is assumed that the control plane is running Kubernetes version N and to be upgraded to version N + 1. Although it is possible to migrate within the same version, ideally the migration should be performed as part of an upgrade so that changes of configuration can be aligned to each release. The exact versions of N and N + 1 depend on each cloud provider. For example, if a cloud provider builds a `cloud-controller-manager` to work with Kubernetes 1.22, then N can be 1.21 and N + 1 can be 1.22.

The control plane nodes should run `kube-controller-manager` with Leader Election enabled through `--leader-elect=true`. As of version N, an in-tree cloud privider must be set with `--cloud-provider` flag and `cloud-controller-manager` should not yet be deployed.

The out-of-tree cloud provider must have built a `cloud-controller-manager` with Leader Migration implementation. If the cloud provider imports `k8s.io/cloud-provider` and `k8s.io/controller-manager` of version v0.21.0 or later, Leader Migration will be available. However, for version before v0.22.0, Leader Migration is alpha and requires feature gate `ControllerManagerLeaderMigration` to be enabled.

This guide assumes that kubelet of each control plane node starts `kube-controller-manager` and `cloud-controller-manager` as static pods defined by their manifests. If the components run in a different setting, please adjust the steps accordingly.

For authorization, this guide assumes that the cluster uses RBAC. If another authorization mode grants permissions to `kube-controller-manager` and `cloud-controller-manager` components, please grant the needed access in a way that matches the mode.

Grant access to Migration Lease

The default permissions of the controller manager allow only accesses to their main Lease. In order for the migration to work, accesses to another Lease are required.

You can grant `kube-controller-manager` full access to the leases API by modifying the `system::leader-locking-kube-controller-manager` role.

This task guide assumes that the name of the migration lease is `cloud-provider-extraction-migration`.

```
kubectl patch -n kube-system role 'system::leader-locking-kube-controller-manager' -p '{"rules": [ {"apiGroups": [ "coordination.k8s.io"], "resources": ["leases"], "resourceNames": ["cloud-provider-extraction-migration"], "verbs": ["create", "list", "get", "update"] } ]}' --type=merge
```

Do the same to the `system::leader-locking-cloud-controller-manager` role.

```
kubectl patch -n kube-system role 'system::leader-locking-cloud-controller-manager' -p '{"rules": [ {"apiGroups": [ "coordination.k8s.io"], "resources": ["leases"], "resourceNames": ["cloud-provider-extraction-migration"], "verbs": ["create", "list", "get", "update"] } ]}' --type=merge
```

Initial Leader Migration configuration

Leader Migration optionally takes a configuration file representing the state of controller-to-manager assignment. At this moment, with in-tree cloud provider, `kube-controller-manager` runs `route`, `service`, and `cloud-node-lifecycle`. The following example configuration shows the assignment.

Leader Migration can be enabled without a configuration. Please see Default Configuration for details.

```yaml
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1beta1
leaderName: cloud-provider-extraction-migration
resourceLock: leases
controllerLeaders:
  - name: route
    component: kube-controller-manager
  - name: service
    component: kube-controller-manager
  - name: cloud-node-lifecycle
    component: kube-controller-manager
```

On each control plane node, save the content to `/etc/leadermigration.conf`, and update the manifest of `kube-controller-manager` so that the file is mounted inside the container at the same location. Also, update the same manifest to add the following arguments:

- `--enable-leader-migration` to enable Leader Migration on the controller manager
- `--leader-migration-config=/etc/leadermigration.conf` to set configuration file

Restart `kube-controller-manager` on each node. At this moment, `kube-controller-manager` has leader migration enabled and is ready for the migration.

Deploy Cloud Controller Manager

In version N + 1, the desired state of controller-to-manager assignment can be represented by a new configuration file, shown as follows. Please note `component` field of each `controllerLeaders` changing from `kube-controller-manager` to `cloud-controller-manager`.

```yaml
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1beta1
leaderName: cloud-provider-extraction-migration
resourceLock: leases
controllerLeaders:
  - name: route
    component: cloud-controller-manager
  - name: service
    component: cloud-controller-manager
  - name: cloud-node-lifecycle
    component: cloud-controller-manager
```

When creating control plane nodes of version N + 1, the content should be deploy to `/etc/leadermigration.conf`. The manifest of `cloud-controller-manager` should be updated to mount the configuration file in the same manner as `kube-controller-manager` of version N. Similarly, add `--feature-gates=ControllerManagerLeaderMigration=true`,`--enable-leader-migration`, and `--leader-migration-config=/etc/leadermigration.conf` to the arguments of `cloud-controller-manager`.

Create a new control plane node of version N + 1 with the updated `cloud-controller-manager` manifest, and with the `--cloud-provider` flag unset for `kube-controller-manager`. `kube-controller-manager` of version N + 1 MUST NOT have Leader Migration enabled because, with an external cloud provider, it does not run the migrated controllers anymore and thus it is not involved in the migration.

Please refer to Cloud Controller Manager Administration for more detail on how to deploy `cloud-controller-manager`.

Upgrade Control Plane

The control plane now contains nodes of both version N and N + 1. The nodes of version N run `kube-controller-manager` only, and these of version N + 1 run both `kube-controller-manager` and `cloud-controller-manager`. The migrated controllers, as specified in the configuration, are running under either `kube-controller-manager` of version N or `cloud-controller-manager` of version N + 1 depending on which controller manager holds the migration lease. No controller will ever be running under both controller managers at any time.

In a rolling manner, create a new control plane node of version N + 1 and bring down one of version N + 1 until the control plane contains only nodes of version N + 1. If a rollback from version N + 1 to N is required, add nodes of version N with Leader Migration enabled for `kube-controller-`

`manager` back to the control plane, replacing one of version N + 1 each time until there are only nodes of version N.

(Optional) Disable Leader Migration

Now that the control plane has been upgraded to run both `kube-controller-manager` and `cloud-controller-manager` of version N + 1, Leader Migration has finished its job and can be safely disabled to save one Lease resource. It is safe to re-enable Leader Migration for the rollback in the future.

In a rolling manager, update manifest of `cloud-controller-manager` to unset both `--enable-leader-migration` and `--leader-migration-config=` flag, also remove the mount of `/etc/leadermigration.conf`, and finally remove `/etc/leadermigration.conf`. To re-enable Leader Migration, recreate the configuration file and add its mount and the flags that enable Leader Migration back to `cloud-controller-manager`.

Default Configuration

Starting Kubernetes 1.22, Leader Migration provides a default configuration suitable for the default controller-to-manager assignment. The default configuration can be enabled by setting `--enable-leader-migration` but without `--leader-migration-config=`.

For `kube-controller-manager` and `cloud-controller-manager`, if there are no flags that enable any in-tree cloud provider or change ownership of controllers, the default configuration can be used to avoid manual creation of the configuration file.

What's next

- Read the Controller Manager Leader Migration enhancement proposal

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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

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 inspect the available namespaces by doing the following:

```
kubectl get namespaces
```

```
NAME      STATUS    AGE
default   Active    13m
```

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:

admin/namespace-dev.json

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "development",
    "labels": {
      "name": "development"
    }
  }
}
```

Create the `development` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-
dev.json
```

Save the following contents into file namespace-prod.json which describes a `production` namespace:

admin/namespace-prod.json

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "production",
    "labels": {
      "name": "production"
    }
  }
}
```

And then let's create the `production` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/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
```

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-92103_kubernetes \
  --user=lithe-cocoa-92103_kubernetes

kubectl config set-context prod --namespace=production \
  --cluster=lithe-cocoa-92103_kubernetes \
  --user=lithe-cocoa-92103_kubernetes
```

By default, the above commands adds two contexts that are saved into file `.kube/config`. You can now view the contexts and alternate against the two new request contexts depending on which namespace you wish to work against.

To view the new contexts:

```
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
- context:
    cluster: lithe-cocoa-92103_kubernetes
    namespace: development
    user: lithe-cocoa-92103_kubernetes
  name: dev
- context:
    cluster: lithe-cocoa-92103_kubernetes
    namespace: production
    user: lithe-cocoa-92103_kubernetes
  name: prod
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
```

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.

admin/snowflake-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: snowflake
  name: snowflake
spec:
  replicas: 2
  selector:
    matchLabels:
      app: snowflake
  template:
    metadata:
      labels:
        app: snowflake
    spec:
      containers:
      - image: k8s.gcr.io/serve_hostname
        imagePullPolicy: Always
        name: snowflake
```

Apply the manifest to create a Deployment

```
kubectl apply -f https://k8s.io/examples/admin/snowflake-deployment.yaml
```

We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment
```

```
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
snowflake     2/2      2             2            2m
```

```
kubectl get pods -l app=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 create deployment cattle --image=k8s.gcr.io/
serve_hostname --replicas=5

kubectl get deployment
```

```
NAME          READY    UP-TO-DATE    AVAILABLE    AGE
cattle        5/5      5             5            10s
```

```
kubectl get pods -l app=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.

Operating etcd clusters for Kubernetes

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

You can find in-depth information about etcd in the official documentation.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 I/O. 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 etcd clusters stable is critical to the stability of Kubernetes clusters. Therefore, run etcd clusters on dedicated machines or isolated environments for guaranteed resource requirements.

- The minimum recommended version of etcd to run in production is `3.2.10+`.

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.

Starting etcd clusters

This section covers starting a single-node and multi-node etcd cluster.

Single-node etcd cluster

Use a single-node etcd cluster only for testing purpose.

1. Run the following:

   ```
   etcd --listen-client-urls=http://$PRIVATE_IP:2379 \
      --advertise-client-urls=http://$PRIVATE_IP:2379
   ```

2. Start the Kubernetes API server with the flag `--etcd-servers=$PRIVATE_IP:2379`.

   Make sure `PRIVATE_IP` is set to 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 --listen-client-urls=http://$IP1:2379,http://$IP2:
   2379,http://$IP3:2379,http://$IP4:2379,http://$IP5:2379 --
   advertise-client-urls=http://$IP1:2379,http://$IP2:
   2379,http://$IP3:2379,http://$IP4:2379,http://$IP5:2379
   ```

2. Start the Kubernetes API servers with the flag `--etcd-servers=$IP1:2379,$IP2:2379,$IP3:2379,$IP4:2379,$IP5:2379`.

   Make sure the `IP<n>` variables are set to 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 the URL schema.

Similarly, to configure etcd with secure client communication, specify flags `--key-file=k8sclient.key` and `--cert-file=k8sclient.cert`, and use HTTPS as the URL schema. Here is an example on a client command that uses secure communication:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  member list
```

Limiting access of etcd clusters

After configuring secure communication, restrict the access of etcd cluster to only the Kubernetes API servers. 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 `--trusted-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 servers the access, configure them with the flags `--etcd-certfile=k8sclient.cert`,`--etcd-keyfile=k8sclient.key` and `--etcd-cafile=ca.cert`.

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://10.0.0.1:2380,
   http://10.0.0.1:2379
   91bc3c398fb3c146, started, member2, http://10.0.0.2:2380,
   http://10.0.0.2:2379
   fd422379fda50e48, started, member3, http://10.0.0.3:2380,
   http://10.0.0.3:2379
   ```

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`:

   ```
   export ETCD_NAME="member4"
   export ETCD_INITIAL_CLUSTER="member2=http://
   10.0.0.2:2380,member3=http://10.0.0.3:2380,member4=http://
   10.0.0.4:2380"
   ```

```
export ETCD_INITIAL_CLUSTER_STATE=existing
etcd [flags]
```

5. Do either of the following:

   1. Update the `--etcd-servers` flag for the Kubernetes API servers to make Kubernetes aware of the configuration changes, then restart the Kubernetes API servers.
   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 control plane 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. 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. Taking the snapshot will 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
```

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.

Snapshot using etcdctl options

We can also take the snapshot using various options given by etcdctl. For example

```
ETCDCTL_API=3 etcdctl -h
```

will list various options available from etcdctl. For example, you can take a snapshot by specifying the endpoint, certificates etc as shown below:

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
  --cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \
  snapshot save <backup-file-location>
```

where `trusted-ca-file`, `cert-file` and `key-file` can be obtained from the description of the etcd Pod.

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. Here is an example:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 snapshot restore
snapshotdb
```

Another example for restoring using etcdctl options:

```
ETCDCTL_API=3 etcdctl --data-dir <data-dir-location> snapshot
restore snapshotdb
```

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 servers 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 servers to fix the issue.

Note:

If any API servers are running in your cluster, you should not attempt to restore instances of etcd. Instead, follow these steps to restore etcd:

- stop *all* API server instances
- restore state in all etcd instances
- restart all API server instances

We also recommend restarting any components (e.g. `kube-scheduler`, `kube-controller-manager`, `kubelet`) to ensure that they don't rely on some stale data. Note that in practice, the restore takes a bit of time. During the restoration, critical components will lose leader lock and restart themselves.

Upgrading etcd clusters

For more details on etcd upgrade, please refer to the etcd upgrades documentation.

Note: Before you start an upgrade, please back up your etcd cluster first.

Reconfigure a Node's Kubelet in a Live Cluster

FEATURE STATE: `Kubernetes v1.22 [deprecated]`

Caution: The Dynamic Kubelet Configuration feature is deprecated and should not be used. Please switch to alternative means distributing configuration to the Nodes of your cluster.

Dynamic Kubelet Configuration allows you to change the configuration of each kubelet in a running Kubernetes cluster, by deploying a ConfigMap and configuring each Node to use it.

Warning: All kubelet configuration parameters can be changed dynamically, but this is unsafe for some parameters. Before deciding to change a parameter dynamically, you need a strong understanding of how that change will affect your cluster's behavior. Always carefully test configuration changes on a small set of nodes before rolling them out cluster-wide. Advice on configuring specific fields is available in the inline `KubeletConfiguration`.

Before you begin

You need to have a Kubernetes cluster. You also need `kubectl`, installed and configured to communicate with your cluster. Make sure that you are using a version of `kubectl` that is compatible with your cluster. Your Kubernetes server must be at or later than version v1.11. To check the version, enter `kubectl version`.

Some of the examples use the command line tool jq. You do not need `jq` to complete the task, because there are manual alternatives.

For each node that you're reconfiguring, you must set the kubelet `--dynamic-config-dir` flag to a writable directory.

Reconfiguring the kubelet on a running 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 corresponding 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, updates a local reference to refer to the file, and exits. For the feature to work correctly, you must be running an OS-level service manager (such as systemd), which will restart the kubelet if it exits. When the kubelet is restarted, it will begin using the new configuration.

The new configuration completely overrides configuration provided by `--config`, and is overridden by command-line flags. Unspecified values in the new configuration will receive default values appropriate to the configuration version (e.g. `kubelet.config.k8s.io/v1beta1`), unless overridden by flags.

The status of the Node's kubelet configuration is reported via `Node.Status.Config`. Once you have updated a Node to use the new ConfigMap, you can observe this status to confirm that the Node is using the intended configuration.

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.

Warning: While it is *possible* to change the configuration by updating the ConfigMap in-place, this causes all kubelets configured with that ConfigMap to update simultaneously. It is much safer to treat ConfigMaps as immutable by convention, aided by `kubectl`'s `--append-hash` option, and incrementally roll out updates to `Node.Spec.ConfigSource`.

Automatic RBAC rules for Node Authorizer

Previously, you were required to manually create RBAC rules to allow Nodes to access their assigned ConfigMaps. The Node Authorizer now automatically configures these rules.

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.

The kubelet loads settings from its configuration file, but you can set command line flags to override the configuration in the file. This means that if you only know the contents of the configuration file, and you don't know the command line overrides, then you do not know the running configuration either.

Because you need to know the running configuration in order to override it, you can fetch the running configuration from the kubelet. You can generate a config file containing a Node's current configuration by accessing the kubelet's `configz` endpoint, through `kubectl proxy`. The next section explains how to do this.

Caution: The kubelet's `configz` endpoint is there to help with debugging, and is not a stable part of kubelet behavior. Do not rely on the behavior of this endpoint for production scenarios or for use with automated tools.

For more information on configuring the kubelet via a configuration file, see Set kubelet parameters via a config file).

Generate the configuration file

Note: The steps below use the `jq` command to streamline working with JSON. To follow the tasks as written, you need to have `jq` installed. You can adapt the steps if you prefer to extract the `kubeletconfig` subobject manually.

1. Choose a Node to reconfigure. In this example, the name of this Node is referred to as `NODE_NAME`.

2. Start the kubectl proxy in the background using the following command:

   ```
   kubectl proxy --port=8001 &
   ```

3. Run the following command to download and unpack the configuration from the `configz` endpoint. The command is long, so be careful when copying and pasting. **If you use zsh**, note that common zsh configurations add backslashes to escape the opening and closing curly braces around the variable name in the URL. For example: `${NODE_NAME}` will be rewritten as `$\{NODE_NAME\}` during the paste. You must remove the backslashes before running the command, or the command will fail.

   ```
   NODE_NAME="the-name-of-the-node-you-are-reconfiguring"; curl -sSL "http://localhost:8001/api/v1/nodes/${NODE_NAME}/proxy/configz" | jq '.kubeletconfig|.kind="KubeletConfiguration"|.apiVersion="kubelet.config.k8s.io/v1beta1"' > kubelet_configz_${NODE_NAME}
   ```

Note: You need to manually add the `kind` and `apiVersion` to the downloaded object, because those fields are not reported by the `configz` endpoint.

Edit the configuration file

Using a text editor, change one of the parameters in the file generated by the previous procedure. For example, you might edit the parameter `eventRecordQPS`, that controls rate limiting for event recording.

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=kubelet_configz_${NODE_NAME} --append-hash -o yaml
```

This is an example of a valid response:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-09-14T20:23:33Z
  name: my-node-config-gkt4c2m4b2
  namespace: kube-system
  resourceVersion: "119980"
  uid: 946d785e-998a-11e7-a8dd-42010a800006
data:
  kubelet: |
    {...}
```

You created that ConfigMap inside the `kube-system` namespace because the kubelet is a Kubernetes system component.

The `--append-hash` option appends a short checksum of the ConfigMap contents to the name. This is convenient for an edit-then-push workflow, because it automatically, yet deterministically, generates new names for new resources. The name that includes this generated hash is referred to as `CONFIG_MAP_NAME` in the following examples.

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}
```

In your text editor, add the following YAML under `spec`:

```
configSource:
    configMap:
        name: CONFIG_MAP_NAME # replace CONFIG_MAP_NAME with the
name of the ConfigMap
        namespace: kube-system
        kubeletConfigKey: kubelet
```

You must specify all three of `name`, `namespace`, and `kubeletConfigKey`. The `kubeletConfigKey` parameter shows the kubelet which key of the ConfigMap contains its config.

Observe that the Node begins using the new configuration

Retrieve the Node using the `kubectl get node ${NODE_NAME} -o yaml` command and inspect `Node.Status.Config`. The config sources corresponding to the `active`, `assigned`, and `lastKnownGood` configurations are reported in the status.

- The `active` configuration is the version the kubelet is currently running with.
- The `assigned` configuration is the latest version the kubelet has resolved based on `Node.Spec.ConfigSource`.
- The `lastKnownGood` configuration is the version the kubelet will fall back to if an invalid config is assigned in `Node.Spec.ConfigSource`.

The`lastKnownGood` configuration might not be present if it is set to its default value, the local config deployed with the node. The status will update `lastKnownGood` to match a valid `assigned` config after the kubelet becomes comfortable with the config. The details of how the kubelet determines a config should become the `lastKnownGood` are not guaranteed by the API, but is currently implemented as a 10-minute grace period.

You can use the following command (using `jq`) to filter down to the config status:

```
kubectl get no ${NODE_NAME} -o json | jq '.status.config'
```

The following is an example response:

```
{
  "active": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  },
  "assigned": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  },
  "lastKnownGood": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  }
}
```

(if you do not have `jq`, you can look at the whole response and find `Node.Status.Config` by eye).

If an error occurs, the kubelet reports it in the `Node.Status.Config.Error` structure. Possible errors are listed in Understanding Node.Status.Config.Error messages. You can search for the identical text in the kubelet log for additional details and context about the error.

Make more changes

Follow the workflow above to make more changes and push them again. Each time you push a ConfigMap with new contents, the `--append-hash` kubectl option creates the ConfigMap with a new name. The safest rollout strategy is to first create a new ConfigMap, and then update the Node to use the new ConfigMap.

Reset the Node to use its local default configuration

To reset the Node to use the configuration it was provisioned with, edit the Node using `kubectl edit node ${NODE_NAME}` and remove the `Node.Spec.ConfigSource` field.

Observe that the Node is using its local default configuration

After removing this subfield, `Node.Status.Config` eventually becomes empty, since all config sources have been reset to `nil`, which indicates that the local default config is `assigned`, `active`, and `lastKnownGood`, and no error is reported.

`kubectl patch` example

You can change a Node's configSource using several different mechanisms. This example uses `kubectl patch`:

```
kubectl patch node ${NODE_NAME} -p "{\"spec\":{\"configSource\":
{\"configMap\":{\"name\":\"${CONFIG_MAP_NAME}\",\"namespace\":
\"kube-system\",\"kubeletConfigKey\":\"kubelet\"}}}}"
```

Understanding how the kubelet checkpoints config

When a new config is assigned to the Node, the kubelet downloads and unpacks the config payload as a set of files on the local disk. The kubelet also records metadata that locally tracks the assigned and last-known-good config sources, so that the kubelet knows which config to use across restarts, even if the API server becomes unavailable. After checkpointing a config and the relevant metadata, the kubelet exits if it detects that the assigned config has changed. When the kubelet is restarted by the OS-level service manager (such as `systemd`), it reads the new metadata and uses the new config.

The recorded metadata is fully resolved, meaning that it contains all necessary information to choose a specific config version - typically a `UID` and `ResourceVersion`. This is in contrast to `Node.Spec.ConfigSource`, where the intended config is declared via the idempotent `namespace/name` that identifies the target ConfigMap; the kubelet tries to use the latest version of this ConfigMap.

When you are debugging problems on a node, you can inspect the kubelet's config metadata and checkpoints. The structure of the kubelet's checkpointing directory is:

```
- --dynamic-config-dir (root for managing dynamic config)
| - meta
  | - assigned (encoded kubeletconfig/
v1beta1.SerializedNodeConfigSource object, indicating the
assigned config)
  | - last-known-good (encoded kubeletconfig/
v1beta1.SerializedNodeConfigSource object, indicating the last-
known-good config)
| - checkpoints
  | - uid1 (dir for versions of object identified by uid1)
    | - resourceVersion1 (dir for unpacked files from
resourceVersion1 of object with uid1)
    | - ...
  | - ...
```

Understanding `Node.Status.Config.Error` messages

The following table describes error messages that can occur when using Dynamic Kubelet Config. You can search for the identical text in the Kubelet log for additional details and context about the error.

Error Message	Possible Causes
failed to load config, see Kubelet log for details	The kubelet likely could not parse the downloaded config payload, or encountered a filesystem error attempting to load the payload from disk.
failed to validate config, see Kubelet log for details	The configuration in the payload, combined with any command-line flag overrides, and the sum of feature gates from flags, the config file, and the remote payload, was determined to be invalid by the kubelet.

Error Message	Possible Causes
invalid NodeConfigSource, exactly one subfield must be non-nil, but all were nil	Since Node.Spec.ConfigSource is validated by the API server to contain at least one non-nil subfield, this likely means that the kubelet is older than the API server and does not recognize a newer source type.
failed to sync: failed to download config, see Kubelet log for details	The kubelet could not download the config. It is possible that Node.Spec.ConfigSource could not be resolved to a concrete API object, or that network errors disrupted the download attempt. The kubelet will retry the download when in this error state.
failed to sync: internal failure, see Kubelet log for details	The kubelet encountered some internal problem and failed to update its config as a result. Examples include filesystem errors and reading objects from the internal informer cache.
internal failure, see Kubelet log for details	The kubelet encountered some internal problem while manipulating config, outside of the configuration sync loop.

What's next

- Set kubelet parameters via a config file explains the supported way to configure a kubelet.
- See the reference documentation for Node, including the `configSource` field within the Node's .spec
- Learn more about kubelet configuration by checking the `KubeletConfiguration` reference.

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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version 1.8. To check the version, enter `kubectl version`. Your Kubernetes server must be at or later than version 1.17 to use the kubelet command line option `--reserved-cpus` to set an explicitly reserved CPU list.

Node Allocatable

'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 `containerd` 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][,][pid=1000]`
- **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.

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for kubernetes system daemons.

To optionally enforce `kube-reserved` on kubernetes 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 the design proposal 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=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **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).

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for OS system daemons.

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.

Explicitly Reserved CPU List

FEATURE STATE: `Kubernetes v1.17 [stable]`

Kubelet Flag: `--reserved-cpus=0-3`

`reserved-cpus` is meant to define an explicit CPU set for OS system daemons and kubernetes system daemons. `reserved-cpus` is for systems that do not intend to define separate top level cgroups for OS system daemons and kubernetes system daemons with regard to cpuset resource. If the Kubelet **does not** have `--system-reserved-cgroup` and `--kube-reserved-cgroup`, the explicit cpuset provided by `reserved-cpus` will take precedence over the CPUs defined by `--kube-reserved` and `--system-reserved` options.

This option is specifically designed for Telco/NFV use cases where uncontrolled interrupts/timers may impact the workload performance. you can use this option to define the explicit cpuset for the system/kubernetes daemons as well as the interrupts/timers, so the rest CPUs on the system can be used exclusively for workloads, with less impact from uncontrolled interrupts/timers. To move the system daemon, kubernetes daemons and interrupts/timers to the explicit cpuset defined by this option, other mechanism outside Kubernetes should be used. For example: in Centos, you can do this using the tuned toolset.

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 on the node pressure eviction page. 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, OOM killed, or unable to fork 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 `88Gi` 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 across pods exceeds 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.

Running Kubernetes Node Components as a Non-root User

FEATURE STATE: `Kubernetes v1.22 [alpha]`

This document describes how to run Kubernetes Node components such as kubelet, CRI, OCI, and CNI without root privileges, by using a user namespace.

This technique is also known as *rootless mode*.

Note:

This document describes how to run Kubernetes Node components (and hence pods) as a non-root user.

If you are just looking for how to run a pod as a non-root user, see SecurityContext.

Before you begin

Your Kubernetes server must be at or later than version 1.22. To check the version, enter `kubectl version`.

- Enable Cgroup v2
- Enable systemd with user session
- Configure several sysctl values, depending on host Linux distribution
- Ensure that your unprivileged user is listed in `/etc/subuid` and `/etc/subgid`
- Enable the `KubeletInUserNamespace` feature gate

Running Kubernetes inside Rootless Docker/Podman

kind

kind supports running Kubernetes inside Rootless Docker or Rootless Podman.

See Running kind with Rootless Docker.

minikube

minikube also supports running Kubernetes inside Rootless Docker.

See the page about the docker driver in the Minikube documentation.

Rootless Podman is not supported.

Running Kubernetes inside Unprivileged Containers

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

sysbox

Sysbox is an open-source container runtime (similar to "runc") that supports running system-level workloads such as Docker and Kubernetes inside unprivileged containers isolated with the Linux user namespace.

See Sysbox Quick Start Guide: Kubernetes-in-Docker for more info.

Sysbox supports running Kubernetes inside unprivileged containers without requiring Cgroup v2 and without the `KubeletInUserNamespace` feature gate. It does this by exposing specially crafted `/proc` and `/sys` filesystems inside the container plus several other advanced OS virtualization techniques.

Running Rootless Kubernetes directly on a host

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

K3s

K3s experimentally supports rootless mode.

See Running K3s with Rootless mode for the usage.

Usernetes

Usernetes is a reference distribution of Kubernetes that can be installed under `$HOME` directory without the root privilege.

Usernetes supports both containerd and CRI-O as CRI runtimes. Usernetes supports multi-node clusters using Flannel (VXLAN).

See the Usernetes repo for the usage.

Manually deploy a node that runs the kubelet in a user namespace

This section provides hints for running Kubernetes in a user namespace manually.

Note: This section is intended to be read by developers of Kubernetes distributions, not by end users.

Creating a user namespace

The first step is to create a user namespace.

If you are trying to run Kubernetes in a user-namespaced container such as Rootless Docker/Podman or LXC/LXD, you are all set, and you can go to the next subsection.

Otherwise you have to create a user namespace by yourself, by calling `unshare(2)` with `CLONE_NEWUSER`.

A user namespace can be also unshared by using command line tools such as:

- `unshare(1)`
- RootlessKit
- become-root

After unsharing the user namespace, you will also have to unshare other namespaces such as mount namespace.

You do *not* need to call `chroot()` nor `pivot_root()` after unsharing the mount namespace, however, you have to mount writable filesystems on several directories *in* the namespace.

At least, the following directories need to be writable *in* the namespace (not *outside* the namespace):

- `/etc`
- `/run`
- `/var/logs`
- `/var/lib/kubelet`
- `/var/lib/cni`
- `/var/lib/containerd` (for containerd)
- `/var/lib/containers` (for CRI-O)

Creating a delegated cgroup tree

In addition to the user namespace, you also need to have a writable cgroup tree with cgroup v2.

Note: Kubernetes support for running Node components in user namespaces requires cgroup v2. Cgroup v1 is not supported.

If you are trying to run Kubernetes in Rootless Docker/Podman or LXC/LXD on a systemd-based host, you are all set.

Otherwise you have to create a systemd unit with `Delegate=yes` property to delegate a cgroup tree with writable permission.

On your node, systemd must already be configured to allow delegation; for more details, see cgroup v2 in the Rootless Containers documentation.

Configuring network

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

The network namespace of the Node components has to have a non-loopback interface, which can be for example configured with slirp4netns, VPNKit, or lxc-user-nic(1).

The network namespaces of the Pods can be configured with regular CNI plugins. For multi-node networking, Flannel (VXLAN, 8472/UDP) is known to work.

Ports such as the kubelet port (10250/TCP) and `NodePort` service ports have to be exposed from the Node network namespace to the host with an external port forwarder, such as RootlessKit, slirp4netns, or socat(1).

You can use the port forwarder from K3s. See Running K3s in Rootless Mode for more details. The implementation can be found in the `pkg/rootlessports` package of k3s.

Configuring CRI

The kubelet relies on a container runtime. You should deploy a container runtime such as containerd or CRI-O and ensure that it is running within the user namespace before the kubelet starts.

- containerd
- CRI-O

Running CRI plugin of containerd in a user namespace is supported since containerd 1.4.

Running containerd within a user namespace requires the following configurations.

```
version = 2

[plugins."io.containerd.grpc.v1.cri"]
# Disable AppArmor
  disable_apparmor = true
# Ignore an error during setting oom_score_adj
  restrict_oom_score_adj = true
# Disable hugetlb cgroup v2 controller (because systemd does not support delegating hugetlb controller)
  disable_hugetlb_controller = true

[plugins."io.containerd.grpc.v1.cri".containerd]
# Using non-fuse overlayfs is also possible for kernel >= 5.11, but requires SELinux to be disabled
  snapshotter = "fuse-overlayfs"

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
# We use cgroupfs that is delegated by systemd, so we do not use SystemdCgroup driver
# (unless you run another systemd in the namespace)
  SystemdCgroup = false
```

The default path of the configuration file is `/etc/containerd/config.toml`. The path can be specified with `containerd -c /path/to/containerd/config.toml`.

Running CRI-O in a user namespace is supported since CRI-O 1.22.

CRI-O requires an environment variable `_CRIO_ROOTLESS=1` to be set.

The following configurations are also recommended:

```
[crio]
  storage_driver = "overlay"
# Using non-fuse overlayfs is also possible for kernel >= 5.11,
but requires SELinux to be disabled
  storage_option = ["overlay.mount_program=/usr/local/bin/fuse-
overlayfs"]

[crio.runtime]
# We use cgroupfs that is delegated by systemd, so we do not use
"systemd" driver
# (unless you run another systemd in the namespace)
  cgroup_manager = "cgroupfs"
```

The default path of the configuration file is `/etc/crio/crio.conf`. The path can be specified with `crio --config /path/to/crio/crio.conf`.

Configuring kubelet

Running kubelet in a user namespace requires the following configuration:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
featureGates:
  KubeletInUserNamespace: true
# We use cgroupfs that is delegated by systemd, so we do not use
"systemd" driver
# (unless you run another systemd in the namespace)
cgroupDriver: "cgroupfs"
```

When the `KubeletInUserNamespace` feature gate is enabled, the kubelet ignores errors that may happen during setting the following sysctl values on the node.

- `vm.overcommit_memory`
- `vm.panic_on_oom`
- `kernel.panic`
- `kernel.panic_on_oops`
- `kernel.keys.root_maxkeys`
- `kernel.keys.root_maxbytes`.

Within a user namespace, the kubelet also ignores any error raised from trying to open `/dev/kmsg`. This feature gate also allows kube-proxy to ignore an error during setting `RLIMIT_NOFILE`.

The `KubeletInUserNamespace` feature gate was introduced in Kubernetes v1.22 with "alpha" status.

Running kubelet in a user namespace without using this feature gate is also possible by mounting a specially crafted proc filesystem (as done by Sysbox), but not officially supported.

Configuring kube-proxy

Running kube-proxy in a user namespace requires the following configuration:

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "iptables" # or "userspace"
conntrack:
# Skip setting sysctl value "net.netfilter.nf_conntrack_max"
  maxPerCore: 0
# Skip setting
"net.netfilter.nf_conntrack_tcp_timeout_established"
  tcpEstablishedTimeout: 0s
# Skip setting "net.netfilter.nf_conntrack_tcp_timeout_close"
  tcpCloseWaitTimeout: 0s
```

Caveats

- Most of "non-local" volume drivers such as `nfs` and `iscsi` do not work. Local volumes like `local`, `hostPath`, `emptyDir`, `configMap`, `secret`, and `downwardAPI` are known to work.
- Some CNI plugins may not work. Flannel (VXLAN) is known to work.

For more on this, see the Caveats and Future work page on the rootlesscontaine.rs website.

See Also

- rootlesscontaine.rs
- Rootless Containers 2020 (KubeCon NA 2020)
- Running kind with Rootless Docker
- Usernetes
- Running K3s with rootless mode
- KEP-2033: Kubelet-in-UserNS (aka Rootless mode)

Safely Drain a Node

This page shows how to safely drain a node, optionally respecting the PodDisruptionBudget you have defined.

Before you begin

Your Kubernetes server must be at or later than version 1.5. To check the version, enter `kubectl version`.

This task also assumes that you have met the following prerequisites:

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 have configured PodDisruptionBudgets for applications that need them.

(Optional) Configure a disruption budget

To ensure that your workloads remain available during maintenance, you can configure a PodDisruptionBudget.

If availability is important for any applications that run or could run on the node(s) that you are draining, configure a PodDisruptionBudgets first and then continue following this guide.

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` ignores 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 respecting the PodDisruptionBudget you have defined). 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 nodes 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` only evicts a pod from the StatefulSet if all three replicas pods are ready; if then you issue multiple drain commands in parallel, Kubernetes respects the PodDisruptionBudget and ensure that only 1 (calculated as `replicas - minAvailable`) 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.

For more information, see API-initiated eviction.

What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 Layer 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 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-scoped 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 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 access to the Kubelet

Kubelets expose HTTPS endpoints which grant powerful control over the node and containers. By default Kubelets allow unauthenticated access to this API.

Production clusters should enable Kubelet authentication and authorization.

Consult the Kubelet authentication/authorization reference 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 run 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.

You can configure Pod security admission to enforce use of a particular Pod Security Standard in a namespace, or to detect breaches.

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 apply the **Baseline** or **Restricted** Pod Security Standard.

Preventing containers from loading unwanted kernel modules

The Linux kernel automatically loads kernel modules from disk if needed in certain circumstances, such as when a piece of hardware is attached or a filesystem is mounted. Of particular relevance to Kubernetes, even unprivileged processes can cause certain network-protocol-related kernel modules to be loaded, just by creating a socket of the appropriate type. This may allow an attacker to exploit a security hole in a kernel module that the administrator assumed was not in use.

To prevent specific modules from being automatically loaded, you can uninstall them from the node, or add rules to block them. On most Linux distributions, you can do that by creating a file such as `/etc/modprobe.d/kubernetes-blacklist.conf` with contents like:

```
# DCCP is unlikely to be needed, has had multiple serious
# vulnerabilities, and is not well-maintained.
blacklist dccp

# SCTP is not used in most Kubernetes clusters, and has also had
# vulnerabilities in the past.
blacklist sctp
```

To block module loading more generically, you can use a Linux Security Module (such as SELinux) to completely deny the `module_request` permission to containers, preventing the kernel from loading modules for containers under any circumstances. (Pods would still be able to use modules that had been loaded manually, or modules that were loaded by the kernel on behalf of some more-privileged process.)

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.

Restricting cloud metadata API access

Cloud platforms (AWS, Azure, GCE, etc.) often expose metadata services locally to instances. By default these APIs are accessible by pods running on an instance and can contain cloud credentials for that node, or provisioning data such as kubelet credentials. These credentials can be used to escalate within the cluster or to other cloud services under the same account.

When running Kubernetes on a cloud platform, limit permissions given to instance credentials, use network policies to restrict pod access to the metadata API, and avoid using provisioning data to deliver secrets.

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 a beta 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 PodSecurityPolicies.

If you use Pod Security admission and allow any component to create Pods within a namespace that permits privileged Pods, those Pods may be able to escape their containers and use this widened access to elevate their privileges.

You should not allow untrusted components to create Pods in any system namespace (those with names that start with `kube-`) nor in any namespace where that access grant allows the possibility of privilege escalation.

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 supports encryption at rest, a feature introduced in 1.7, and beta since 1.13. This 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 beta, it offers 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.

Set Kubelet parameters via a config file

A subset of the Kubelet's configuration parameters may be set via an on-disk config file, as a substitute for command-line flags.

Providing parameters via a config file is the recommended approach because it simplifies node deployment and configuration management.

Create the config file

The subset of the Kubelet's configuration that can be configured via a file is defined by the `KubeletConfiguration` struct.

The configuration file must be a JSON or YAML representation of the parameters in this struct. Make sure the Kubelet has read permissions on the file.

Here is an example of what this file might look like:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
address: "192.168.0.8",
port: 20250,
serializeImagePulls: false,
evictionHard:
    memory.available:  "200Mi"
```

In the example, the Kubelet is configured to serve on IP address 192.168.0.8 and port 20250, pull images in parallel, and evict Pods when available memory drops below 200Mi. All other Kubelet configuration values are left at their built-in defaults, unless overridden by flags. Command line flags which target the same value as a config file will override that value.

Start a Kubelet process configured via the config file

Note: If you use kubeadm to initialize your cluster, use the kubelet-config while creating your cluster with `kubeadmin init`. See configuring kubelet using kubeadm for details.

Start the Kubelet with the `--config` flag set to the path of the Kubelet's config file. The Kubelet will then load its config from this file.

Note that command line flags which target the same value as a config file will override that value. This helps ensure backwards compatibility with the command-line API.

Note that relative file paths in the Kubelet config file are resolved relative to the location of the Kubelet config file, whereas relative paths in command line flags are resolved relative to the Kubelet's current working directory.

Note that some default values differ between command-line flags and the Kubelet config file. If `--config` is provided and the values are not specified via the command line, the defaults for the `KubeletConfiguration` version apply. In the above example, this version is `kubelet.config.k8s.io/v1beta1`.

What's next

- Learn more about kubelet configuration by checking the `KubeletConfiguration` reference.

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.
- You 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
kube-public   Active    11d
```

Kubernetes starts with three initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system
- `kube-public` This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

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

For more details, see Namespace in the API reference.

Creating a new namespace

Note: Avoid creating namespace with prefix `kube-`, since it is reserved for Kubernetes system namespaces.

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
   ```

2. Alternatively, you can create namespace using below command:

   ```
   kubectl create namespace <insert-namespace-name-here>
   ```

The name of your namespace must be a valid DNS 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

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 namespaces by doing the following:

   ```
   kubectl get namespaces
   ```

   ```
   NAME      STATUS    AGE
   default   Active    13m
   ```

2. 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.

Create the `development` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/namespace-
dev.json
```

And then let's create the `production` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/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
```

3. 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.

```
kubectl create deployment snowflake --image=k8s.gcr.io/
serve_hostname  -n=development --replicas=2
```

   We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment -n=development
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
snowflake    2/2     2            2           2m
```

```
kubectl get pods -l app=snowflake -n=development
```

```
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.

The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment -n=production
kubectl get pods -n=production
```

Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=k8s.gcr.io/
serve_hostname -n=production
kubectl scale deployment cattle --replicas=5 -n=production

kubectl get deployment -n=production
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
cattle       5/5     5            5           10s
```

```
kubectl get pods -l app=cattle -n=production
```

```
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 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.

Upgrade A Cluster

This page provides an overview of the steps you should follow to upgrade a Kubernetes cluster.

The way that you upgrade a cluster depends on how you initially deployed it and on any subsequent changes.

At a high level, the steps you perform are:

- Upgrade the control plane
- Upgrade the nodes in your cluster
- Upgrade clients such as kubectl
- Adjust manifests and other resources based on the API changes that accompany the new Kubernetes version

Before you begin

You must have an existing cluster. This page is about upgrading from Kubernetes 1.22 to Kubernetes 1.23. If your cluster is not currently running Kubernetes 1.22 then please check the documentation for the version of Kubernetes that you plan to upgrade to.

Upgrade approaches

kubeadm

If your cluster was deployed using the `kubeadm` tool, refer to Upgrading kubeadm clusters for detailed information on how to upgrade the cluster.

Once you have upgraded the cluster, remember to install the latest version of `kubectl`.

Manual deployments

Caution: These steps do not account for third-party extensions such as network and storage plugins.

You should manually update the control plane following this sequence:

- etcd (all instances)
- kube-apiserver (all control plane hosts)
- kube-controller-manager
- kube-scheduler
- cloud controller manager, if you use one

At this point you should install the latest version of `kubectl`.

For each node in your cluster, drain that node and then either replace it with a new node that uses the 1.23 kubelet, or upgrade the kubelet on that node and bring the node back into service.

Other deployments

Refer to the documentation for your cluster deployment tool to learn the recommended set up steps for maintenance.

Post-upgrade tasks

Switch your cluster's storage API version

The objects that are serialized into etcd 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.

For each affected object, fetch it using the latest supported API and then write it back also using the latest supported API.

Update manifests

Upgrading to a new Kubernetes version can provide new APIs.

You can use `kubectl convert` command to convert manifests between different API versions. For example:

```
kubectl convert -f pod.yaml --output-version v1
```

The `kubectl` tool replaces the contents of `pod.yaml` with a manifest that sets `kind` to Pod (unchanged), but with a revised `apiVersion`.

Use Cascading Deletion in a Cluster

This page shows you how to specify the type of cascading deletion to use in your cluster during garbage collection.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as

control plane hosts. 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 also need to create a sample Deployment to experiment with the different types of cascading deletion. You will need to recreate the Deployment for each type.

Check owner references on your pods

Check that the `ownerReferences` field is present on your pods:

```
kubectl get pods -l app=nginx --output=yaml
```

The output has an `ownerReferences` field similar to this:

```
apiVersion: v1
    ...
    ownerReferences:
    - apiVersion: apps/v1
      blockOwnerDeletion: true
      controller: true
      kind: ReplicaSet
      name: nginx-deployment-6b474476c4
      uid: 4fdcd81c-bd5d-41f7-97af-3a3b759af9a7
    ...
```

Use foreground cascading deletion

By default, Kubernetes uses background cascading deletion to delete dependents of an object. You can switch to foreground cascading deletion using either `kubectl` or the Kubernetes API, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

- Kubernetes 1.20.x and later
- Versions prior to Kubernetes 1.20.x

You can delete objects using foreground cascading deletion using `kubectl` or the Kubernetes API.

Using kubectl

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=foreground
```

Using the Kubernetes API

1. Start a local proxy session:

```
kubectl proxy --port=8080
```

2. Use `curl` to trigger deletion:

```
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/
default/deployments/nginx-deployment \
    -d '{"kind":"DeleteOptions","apiVersion":"v1","propagatio
nPolicy":"Foreground"}' \
    -H "Content-Type: application/json"
```

The output contains a `foregroundDeletion` finalizer like this:

```
"kind": "Deployment",
"apiVersion": "apps/v1",
"metadata": {
    "name": "nginx-deployment",
    "namespace": "default",
    "uid": "d1ce1b02-cae8-4288-8a53-30e84d8fa505",
    "resourceVersion": "1363097",
    "creationTimestamp": "2021-07-08T20:24:37Z",
    "deletionTimestamp": "2021-07-08T20:27:39Z",
    "finalizers": [
      "foregroundDeletion"
    ]
    ...
```

You can delete objects using foreground cascading deletion by calling the Kubernetes API.

For details, read the documentation for your Kubernetes version.

1. Start a local proxy session:

```
kubectl proxy --port=8080
```

2. Use `curl` to trigger deletion:

```
curl -X DELETE localhost:8080/apis/apps/v1/namespaces/
default/deployments/nginx-deployment \
    -d '{"kind":"DeleteOptions","apiVersion":"v1","propagatio
nPolicy":"Foreground"}' \
    -H "Content-Type: application/json"
```

The output contains a `foregroundDeletion` finalizer like this:

```
"kind": "Deployment",
"apiVersion": "apps/v1",
"metadata": {
    "name": "nginx-deployment",
    "namespace": "default",
    "uid": "d1ce1b02-cae8-4288-8a53-30e84d8fa505",
    "resourceVersion": "1363097",
    "creationTimestamp": "2021-07-08T20:24:37Z",
    "deletionTimestamp": "2021-07-08T20:27:39Z",
    "finalizers": [
      "foregroundDeletion"
    ]
    ...
```

Use background cascading deletion

1. Create a sample Deployment.
2. Use either `kubectl` or the Kubernetes API to delete the Deployment, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

- Kubernetes version 1.20.x and later
- Versions prior to Kubernetes 1.20.x

You can delete objects using background cascading deletion using `kubectl` or the Kubernetes API.

Kubernetes uses background cascading deletion by default, and does so even if you run the following commands without the `--cascade` flag or the `propagationPolicy` argument.

Using kubectl

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=background
```

Using the Kubernetes API

1. Start a local proxy session:

   ```
   kubectl proxy --port=8080
   ```

2. Use `curl` to trigger deletion:

   ```
   curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/deployments/nginx-deployment \
       -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' \
       -H "Content-Type: application/json"
   ```

   The output is similar to this:

   ```
   "kind": "Status",
   "apiVersion": "v1",
   ...
   "status": "Success",
   "details": {
       "name": "nginx-deployment",
       "group": "apps",
       "kind": "deployments",
   ```

```
    "uid": "cc9eefb9-2d49-4445-b1c1-d261c9396456"
}
```

Kubernetes uses background cascading deletion by default, and does so even if you run the following commands without the `--cascade` flag or the `propagationPolicy: Background` argument.

For details, read the documentation for your Kubernetes version.

Using kubectl

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=true
```

Using the Kubernetes API

1. Start a local proxy session:

   ```
   kubectl proxy --port=8080
   ```

2. Use `curl` to trigger deletion:

   ```
   curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/deployments/nginx-deployment \
       -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' \
       -H "Content-Type: application/json"
   ```

   The output is similar to this:

   ```
   "kind": "Status",
   "apiVersion": "v1",
   ...
   "status": "Success",
   "details": {
       "name": "nginx-deployment",
       "group": "apps",
       "kind": "deployments",
       "uid": "cc9eefb9-2d49-4445-b1c1-d261c9396456"
   }
   ```

Delete owner objects and orphan dependents

By default, when you tell Kubernetes to delete an object, the controller also deletes dependent objects. You can make Kubernetes *orphan* these dependents using `kubectl` or the Kubernetes API, depending on the Kubernetes version your cluster runs. To check the version, enter `kubectl version`.

- Kubernetes version 1.20.x and later
- Versions prior to Kubernetes 1.20.x

Using kubectl

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=orphan
```

Using the Kubernetes API

1. Start a local proxy session:

   ```
   kubectl proxy --port=8080
   ```

2. Use `curl` to trigger deletion:

   ```
   curl -X DELETE localhost:8080/apis/apps/v1/namespaces/
   default/deployments/nginx-deployment \
       -d '{"kind":"DeleteOptions","apiVersion":"v1","propagatio
   nPolicy":"Orphan"}' \
       -H "Content-Type: application/json"
   ```

   The output contains `orphan` in the `finalizers` field, similar to this:

   ```
   "kind": "Deployment",
   "apiVersion": "apps/v1",
   "namespace": "default",
   "uid": "6f577034-42a0-479d-be21-78018c466f1f",
   "creationTimestamp": "2021-07-09T16:46:37Z",
   "deletionTimestamp": "2021-07-09T16:47:08Z",
   "deletionGracePeriodSeconds": 0,
   "finalizers": [
     "orphan"
   ],
   ...
   ```

For details, read the documentation for your Kubernetes version.

Using kubectl

Run the following command:

```
kubectl delete deployment nginx-deployment --cascade=orphan
```

Using the Kubernetes API

1. Start a local proxy session:

   ```
   kubectl proxy --port=8080
   ```

2. Use `curl` to trigger deletion:

   ```
   curl -X DELETE localhost:8080/apis/apps/v1/namespaces/
   default/deployments/nginx-deployment \
       -d '{"kind":"DeleteOptions","apiVersion":"v1","propagatio
   ```

```
nPolicy":"Orphan"}' \
    -H "Content-Type: application/json"
```

The output contains `orphan` in the `finalizers` field, similar to this:

```
"kind": "Deployment",
"apiVersion": "apps/v1",
"namespace": "default",
"uid": "6f577034-42a0-479d-be21-78018c466f1f",
"creationTimestamp": "2021-07-09T16:46:37Z",
"deletionTimestamp": "2021-07-09T16:47:08Z",
"deletionGracePeriodSeconds": 0,
"finalizers": [
  "orphan"
],
...
```

You can check that the Pods managed by the Deployment are still running:

```
kubectl get pods -l app=nginx
```

What's next

- Learn about owners and dependents in Kubernetes.
- Learn about Kubernetes finalizers.
- Learn about garbage collection.

Using a KMS provider for data encryption

This page shows how to configure a Key Management Service (KMS) provider and plugin to enable secret data encryption.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

- Kubernetes version 1.10.0 or later is required

- etcd v3 or later is required

FEATURE STATE: `Kubernetes v1.12 [beta]`

The KMS encryption provider uses an envelope encryption scheme to encrypt data in etcd. The data is encrypted using a data encryption key (DEK); a new DEK is generated for each encryption. The DEKs are encrypted with a key encryption key (KEK) that is stored and managed in a remote KMS. The KMS provider uses gRPC to communicate with a specific KMS plugin. The KMS plugin, which is implemented as a gRPC server and deployed on the same host(s) as the Kubernetes master(s), is responsible for all communication with the remote KMS.

Configuring the KMS provider

To configure a KMS provider on the API server, include a provider of type `kms` in the providers array in the encryption configuration file and set the following properties:

- `name`: Display name of the KMS plugin.
- `endpoint`: Listen address of the gRPC server (KMS plugin). The endpoint is a UNIX domain socket.
- `cachesize`: Number of data encryption keys (DEKs) to be cached in the clear. When cached, DEKs can be used without another call to the KMS; whereas DEKs that are not cached require a call to the KMS to unwrap.
- `timeout`: How long should kube-apiserver wait for kms-plugin to respond before returning an error (default is 3 seconds).

See Understanding the encryption at rest configuration.

Implementing a KMS plugin

To implement a KMS plugin, you can develop a new plugin gRPC server or enable a KMS plugin already provided by your cloud provider. You then integrate the plugin with the remote KMS and deploy it on the Kubernetes master.

Enabling the KMS supported by your cloud provider

Refer to your cloud provider for instructions on enabling the cloud provider-specific KMS plugin.

Developing a KMS plugin gRPC server

You can develop a KMS plugin gRPC server using a stub file available for Go. For other languages, you use a proto file to create a stub file that you can use to develop the gRPC server code.

- Using Go: Use the functions and data structures in the stub file: service.pb.go to develop the gRPC server code

- Using languages other than Go: Use the protoc compiler with the proto file: service.proto to generate a stub file for the specific language

Then use the functions and data structures in the stub file to develop the server code.

Notes:

- kms plugin version: `v1beta1`

  In response to procedure call Version, a compatible KMS plugin should return v1beta1 as VersionResponse.version.

- message version: `v1beta1`

  All messages from KMS provider have the version field set to current version v1beta1.

- protocol: UNIX domain socket (`unix`)

  The gRPC server should listen at UNIX domain socket.

Integrating a KMS plugin with the remote KMS

The KMS plugin can communicate with the remote KMS using any protocol supported by the KMS. All configuration data, including authentication credentials the KMS plugin uses to communicate with the remote KMS, are stored and managed by the KMS plugin independently. The KMS plugin can encode the ciphertext with additional metadata that may be required before sending it to the KMS for decryption.

Deploying the KMS plugin

Ensure that the KMS plugin runs on the same host(s) as the Kubernetes master(s).

Encrypting your data with the KMS provider

To encrypt the data:

1. Create a new encryption configuration file using the appropriate properties for the `kms` provider:

   ```
   apiVersion: apiserver.config.k8s.io/v1
   kind: EncryptionConfiguration
   resources:
     - resources:
         - secrets
       providers:
         - kms:
             name: myKmsPlugin
             endpoint: unix:///tmp/socketfile.sock
             cachesize: 100
             timeout: 3s
         - identity: {}
   ```

2. Set the `--encryption-provider-config` flag on the kube-apiserver to point to the location of the configuration file.

3. Restart your API server.

Verifying that the 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 verify, 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 command line, 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:kms:v1:`, which indicates that the `kms` provider has encrypted the resulting data.

4. Verify that the secret is correctly decrypted when retrieved via the API:

   ```
   kubectl describe secret secret1 -n default
   ```

   should match `mykey: mydata`

Ensuring all secrets are encrypted

Because secrets are encrypted on write, performing an update on a secret encrypts that content.

The following command 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.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -
f -
```

Switching from a local encryption provider to the KMS provider

To switch from a local encryption provider to the `kms` provider and re-encrypt all of the secrets:

1. Add the `kms` provider as the first entry in the configuration file as shown in the following example.

   ```
   apiVersion: apiserver.config.k8s.io/v1
   kind: EncryptionConfiguration
   resources:
     - resources:
         - secrets
       providers:
         - kms:
             name : myKmsPlugin
             endpoint: unix:///tmp/socketfile.sock
             cachesize: 100
         - aescbc:
             keys:
               - name: key1
                 secret: <BASE 64 ENCODED SECRET>
   ```

2. Restart all kube-apiserver processes.

3. Run the following command to force all secrets to be re-encrypted using the `kms` provider.

   ```
   kubectl get secrets --all-namespaces -o json| kubectl
   replace -f -
   ```

Disabling encryption at rest

To disable encryption at rest:

1. Place the `identity` provider as the first entry in the configuration file:

   ```
   apiVersion: apiserver.config.k8s.io/v1
   kind: EncryptionConfiguration
   resources:
     - resources:
         - secrets
       providers:
         - identity: {}
         - kms:
             name : myKmsPlugin
             endpoint: unix:///tmp/socketfile.sock
             cachesize: 100
   ```

2. Restart all kube-apiserver processes.

3. Run the following command to force all secrets to be decrypted.

   ```
   kubectl get secrets --all-namespaces -o json | kubectl
   replace -f -
   ```

Using CoreDNS for Service Discovery

This page describes the CoreDNS upgrade process and how to install CoreDNS instead of kube-dns.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.9. To check the version, enter `kubectl version`.

About CoreDNS

CoreDNS is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. Like Kubernetes, the CoreDNS project is hosted by the CNCF.

You can use CoreDNS instead of kube-dns in your cluster by replacing kube-dns in an existing deployment, or by using tools like kubeadm that will deploy and upgrade the cluster for you.

Installing CoreDNS

For manual deployment or replacement of kube-dns, see the documentation at the CoreDNS GitHub project.

Migrating to CoreDNS

Upgrading an existing cluster with kubeadm

In Kubernetes version 1.21, kubeadm removed its support for `kube-dns` as a DNS application. For `kubeadm` v1.23, the only supported cluster DNS application is CoreDNS.

You can move to CoreDNS when you use `kubeadm` to upgrade a cluster that is using `kube-dns`. In this case, `kubeadm` generates the CoreDNS configuration ("Corefile") based upon the `kube-dns` ConfigMap, preserving configurations for stub domains, and upstream name server.

Upgrading CoreDNS

You can check the version of CoreDNS that kubeadm installs for each version of Kubernetes in the page CoreDNS version in Kubernetes.

CoreDNS can be upgraded manually in case you want to only upgrade CoreDNS or use your own custom image. There is a helpful guideline and walkthrough available to ensure a smooth upgrade. Make sure the existing CoreDNS configuration ("Corefile") is retained when upgrading your cluster.

If you are upgrading your cluster using the `kubeadm` tool, `kubeadm` can take care of retaining the existing CoreDNS configuration automatically.

Tuning CoreDNS

When resource utilisation is a concern, it may be useful to tune the configuration of CoreDNS. For more details, check out the documentation on scaling CoreDNS.

What's next

You can configure CoreDNS to support many more use cases than kube-dns does by modifying the CoreDNS configuration ("Corefile"). For more information, see the documentation for the `kubernetes` CoreDNS plugin, or read the Custom DNS Entries for Kubernetes. in the CoreDNS blog.

Using NodeLocal DNSCache in Kubernetes clusters

FEATURE STATE: `Kubernetes v1.18 [stable]`

This page provides an overview of NodeLocal DNSCache feature in Kubernetes.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as

control plane hosts. 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 check the version, enter `kubectl version`.

Introduction

NodeLocal DNSCache improves Cluster DNS performance by running a DNS caching agent on cluster nodes as a DaemonSet. In today's architecture, Pods in 'ClusterFirst' DNS mode reach out to a kube-dns `serviceIP` for DNS queries. This is translated to a kube-dns/CoreDNS endpoint via iptables rules added by kube-proxy. With this new architecture, Pods will reach out to the DNS caching agent running on the same node, thereby avoiding iptables DNAT rules and connection tracking. The local caching agent will query kube-dns service for cache misses of cluster hostnames ("`cluster.local`" suffix by default).

Motivation

- With the current DNS architecture, it is possible that Pods with the highest DNS QPS have to reach out to a different node, if there is no local kube-dns/CoreDNS instance. Having a local cache will help improve the latency in such scenarios.

- Skipping iptables DNAT and connection tracking will help reduce conntrack races and avoid UDP DNS entries filling up conntrack table.

- Connections from local caching agent to kube-dns service can be upgraded to TCP. TCP conntrack entries will be removed on connection close in contrast with UDP entries that have to timeout (default `nf_conntrack_udp_timeout` is 30 seconds)

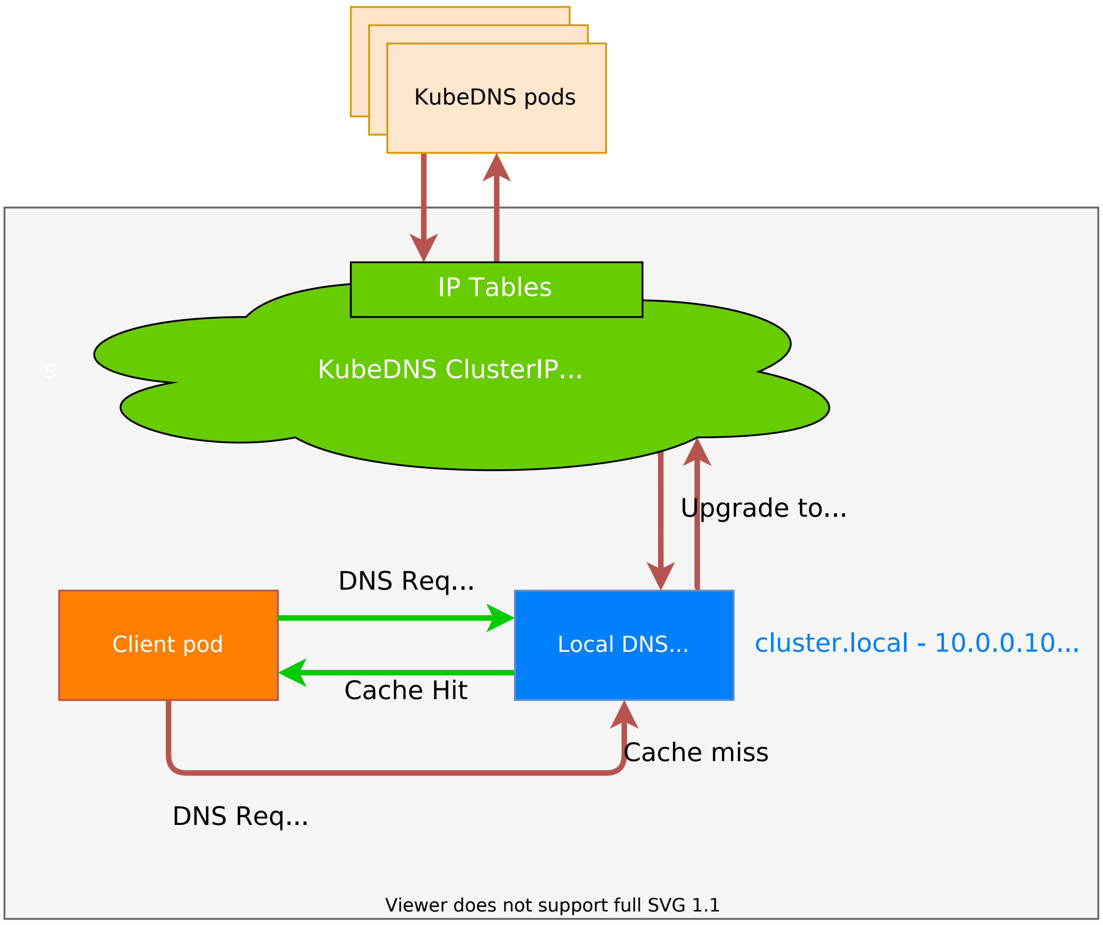
- Upgrading DNS queries from UDP to TCP would reduce tail latency attributed to dropped UDP packets and DNS timeouts usually up to 30s (3 retries + 10s timeout). Since the nodelocal cache listens for UDP DNS queries, applications don't need to be changed.

- Metrics & visibility into DNS requests at a node level.

- Negative caching can be re-enabled, thereby reducing number of queries to kube-dns service.

Architecture Diagram

This is the path followed by DNS Queries after NodeLocal DNSCache is enabled:



Nodelocal DNSCache flow

This image shows how NodeLocal DNSCache handles DNS queries.

Configuration

Note: The local listen IP address for NodeLocal DNSCache can be any address that can be guaranteed to not collide with any existing IP in your cluster. It's recommended to use an address with a local scope, per example, from the 'link-local' range '169.254.0.0/16' for IPv4 or from the 'Unique Local Address' range in IPv6 'fd00::/8'.

This feature can be enabled using the following steps:

- Prepare a manifest similar to the sample `nodelocaldns.yaml` and save it as `nodelocaldns.yaml.`
- If using IPv6, the CoreDNS configuration file need to enclose all the IPv6 addresses into square brackets if used in 'IP:Port' format. If you are using the sample manifest from the previous point, this will require to modify the configuration line L70 like this: "`health [__PILLAR__LOCAL__DNS__]:8080`"
- Substitute the variables in the manifest with the right values:

```
kubedns=`kubectl get svc kube-dns -n kube-system -o jsonpath=
{.spec.clusterIP}`
domain=<cluster-domain>
localdns=<node-local-address>
```

`<cluster-domain>` is "`cluster.local`" by default. `<node-local-address>` is the local listen IP address chosen for NodeLocal DNSCache.

- If kube-proxy is running in IPTABLES mode:

  ```
  sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/
  __PILLAR__DNS__DOMAIN__/$domain/g; s/
  __PILLAR__DNS__SERVER__/$kubedns/g" nodelocaldns.yaml
  ```

  `__PILLAR__CLUSTER__DNS__` and `__PILLAR__UPSTREAM__SERVERS__` will be populated by the `node-local-dns` pods. In this mode, the `node-local-dns` pods listen on both the kube-dns service IP as well as `<node-local-address>`, so pods can lookup DNS records using either IP address.

- If kube-proxy is running in IPVS mode:

  ```
  sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/
  __PILLAR__DNS__DOMAIN__/$domain/g;
  s/,__PILLAR__DNS__SERVER__//g; s/
  __PILLAR__CLUSTER__DNS__/$kubedns/g" nodelocaldns.yaml
  ```

  In this mode, the `node-local-dns` pods listen only on `<node-local-address>`. The `node-local-dns` interface cannot bind the kube-dns cluster IP since the interface used for IPVS loadbalancing already uses this address. `__PILLAR__UPSTREAM__SERVERS__` will be populated by the node-local-dns pods.

- Run `kubectl create -f nodelocaldns.yaml`

- If using kube-proxy in IPVS mode, `--cluster-dns` flag to kubelet needs to be modified to use `<node-local-address>` that NodeLocal DNSCache is listening on. Otherwise, there is no need to modify the value of the `--cluster-dns` flag, since NodeLocal DNSCache listens on both the kube-dns service IP as well as `<node-local-address>`.

Once enabled, the `node-local-dns` Pods will run in the `kube-system` namespace on each of the cluster nodes. This Pod runs CoreDNS in cache mode, so all CoreDNS metrics exposed by the different plugins will be available on a per-node basis.

You can disable this feature by removing the DaemonSet, using `kubectl delete -f <manifest>`. You should also revert any changes you made to the kubelet configuration.

StubDomains and Upstream server Configuration

StubDomains and upstream servers specified in the `kube-dns` ConfigMap in the `kube-system` namespace are automatically picked up by `node-local-dns` pods. The ConfigMap contents need to follow the format shown in the example. The `node-local-dns` ConfigMap can also be modified directly with the stubDomain configuration in the Corefile format. Some cloud providers might not allow modifying `node-local-dns` ConfigMap directly. In those cases, the `kube-dns` ConfigMap can be updated.

Setting memory limits

The `node-local-dns` Pods use memory for storing cache entries and processing queries. Since they do not watch Kubernetes objects, the cluster size or the number of Services/Endpoints do not directly affect memory usage. Memory usage is influenced by the DNS query pattern. From CoreDNS docs,

> The default cache size is 10000 entries, which uses about 30 MB when completely filled.

This would be the memory usage for each server block (if the cache gets completely filled). Memory usage can be reduced by specifying smaller cache sizes.

The number of concurrent queries is linked to the memory demand, because each extra goroutine used for handling a query requires an amount of memory. You can set an upper limit using the `max_concurrent` option in the forward plugin.

If a `node-local-dns` Pod attempts to use more memory than is available (because of total system resources, or because of a configured resource limit), the operating system may shut down that pod's container. If this happens, the container that is terminated ("OOMKilled") does not clean up the custom packet filtering rules that it previously added during startup. The `node-local-dns` container should get restarted (since managed as part of a DaemonSet), but this will lead to a brief DNS downtime each time that the container fails: the packet filtering rules direct DNS queries to a local Pod that is unhealthy.

You can determine a suitable memory limit by running node-local-dns pods without a limit and measuring the peak usage. You can also set up and use a VerticalPodAutoscaler in *recommender mode*, and then check its recommendations.

Using sysctls in a Kubernetes Cluster

FEATURE STATE: `Kubernetes v1.21 [stable]`

This document describes how to configure and use kernel parameters within a Kubernetes cluster using the sysctl interface.

Note: Starting from Kubernetes version 1.23, the kubelet supports the use of either `/` or `.` as separators for sysctl names. For example, you can represent the same sysctl name as `kernel.shm_rmid_forced` using a period as the separator, or as `kernel/shm_rmid_forced` using a slash as a separator. For more sysctl parameter conversion method details, please refer to the page sysctl.d(5) from the Linux man-pages project. Setting Sysctls for a Pod and PodSecurityPolicy features do not yet support setting sysctls with slashes.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

For some steps, you also need to be able to reconfigure the command line options for the kubelets running on your cluster.

Listing all Sysctl Parameters

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
```

Enabling 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*. The following sysctls are supported in the *safe* set:

- `kernel.shm_rmid_forced`,
- `net.ipv4.ip_local_port_range`,
- `net.ipv4.tcp_syncookies`,
- `net.ipv4.ping_group_range` (since Kubernetes 1.18),
- `net.ipv4.ip_unprivileged_port_start` (since Kubernetes 1.22).

Note: The example `net.ipv4.tcp_syncookies` is not namespaced on Linux kernel version 4.4 or lower.

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.

With the warning above in mind, the cluster admin can allow certain *unsafe* sysctls for very special situations such as high-performance or real-time application tuning. *Unsafe* sysctls are enabled on a node-by-node basis with a flag of the kubelet; for example:

```
kubelet --allowed-unsafe-sysctls \
  'kernel.msg*,net.core.somaxconn' ...
```

For Minikube, this can be done via the `extra-config` flag:

```
minikube start --extra-config="kubelet.allowed-unsafe-
sysctls=kernel.msg*,net.core.somaxconn"...
```

Only *namespaced* sysctls can be enabled this way.

Setting Sysctls for a Pod

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. Only namespaced sysctls are configurable via the pod securityContext within Kubernetes.

The following sysctls are known to be namespaced. This list could change in future versions of the Linux kernel.

- `kernel.shm*`,
- `kernel.msg*`,
- `kernel.sem`,
- `fs.mqueue.*`,
- The parameters under `net.*` that can be set in container networking namespace. However, there are exceptions (e.g., `net.netfilter.nf_conntrack_max` and `net.netfilter.nf_conntrack_expect_max` can be set in container networking namespace but they are unnamespaced).

Sysctls with no namespace are called *node-level* sysctls. If you need to set them, you must manually configure them on each node's operating system, or by using a DaemonSet with privileged containers.

Use the pod securityContext to configure namespaced sysctls. The securityContext applies to all containers in the same pod.

This example uses the pod securityContext to set a safe sysctl `kernel.shm_rmid_forced` and two unsafe sysctls `net.core.somaxconn` and `kernel.msgmax`. There is no distinction between *safe* and *unsafe* sysctls in the specification.

Warning: Only modify sysctl parameters after you understand their effects, to avoid destabilizing your operating system.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
  ...
```

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.

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.

A pod with the *unsafe* sysctls 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.

PodSecurityPolicy

FEATURE STATE: `Kubernetes v1.21 [deprecated]`

You can further control which sysctls can be set in pods by specifying lists of sysctls or sysctl patterns in the `forbiddenSysctls` and/or `allowedUnsafeSysctls` fields of the PodSecurityPolicy. A sysctl pattern ends with a `*` character, such as `kernel.*`. A `*` character on its own matches all sysctls.

By default, all safe sysctls are allowed.

Both `forbiddenSysctls` and `allowedUnsafeSysctls` are lists of plain sysctl names or sysctl patterns (which end with `*`). The string `*` matches all sysctls.

The `forbiddenSysctls` field excludes specific sysctls. You can forbid a combination of safe and unsafe sysctls in the list. To forbid setting any sysctls, use `*` on its own.

If you specify any unsafe sysctl in the `allowedUnsafeSysctls` field and it is not present in the `forbiddenSysctls` field, that sysctl can be used in Pods using this PodSecurityPolicy. To allow all unsafe sysctls in the PodSecurityPolicy to be set, use `*` on its own.

Do not configure these two fields such that there is overlap, meaning that a given sysctl is both allowed and forbidden.

Warning: If you allow unsafe sysctls via the `allowedUnsafeSysctls` field in a PodSecurityPolicy, any pod using such a sysctl will fail to start if the sysctl is not allowed via the `--allowed-unsafe-sysctls` kubelet flag as well on that node.

This example allows unsafe sysctls prefixed with `kernel.msg` to be set and disallows setting of the `kernel.shm_rmid_forced` sysctl.

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - kernel.msg*
  forbiddenSysctls:
  - kernel.shm_rmid_forced
 ...
```

Utilizing the NUMA-aware Memory Manager

FEATURE STATE: `Kubernetes v1.22 [beta]`

The Kubernetes *Memory Manager* enables the feature of guaranteed memory (and hugepages) allocation for pods in the `Guaranteed` QoS class.

The Memory Manager employs hint generation protocol to yield the most suitable NUMA affinity for a pod. The Memory Manager feeds the central manager (*Topology Manager*) with these affinity hints. Based on both the hints and Topology Manager policy, the pod is rejected or admitted to the node.

Moreover, the Memory Manager ensures that the memory which a pod requests is allocated from a minimum number of NUMA nodes.

The Memory Manager is only pertinent to Linux based hosts.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

To align memory resources with other requested resources in a Pod spec:

- the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See control CPU Management Policies;
- the Topology Manager should be enabled and proper Topology Manager policy should be configured on a Node. See control Topology Management Policies.

Starting from v1.22, the Memory Manager is enabled by default through `MemoryManager` feature gate.

Preceding v1.22, the `kubelet` must be started with the following flag:

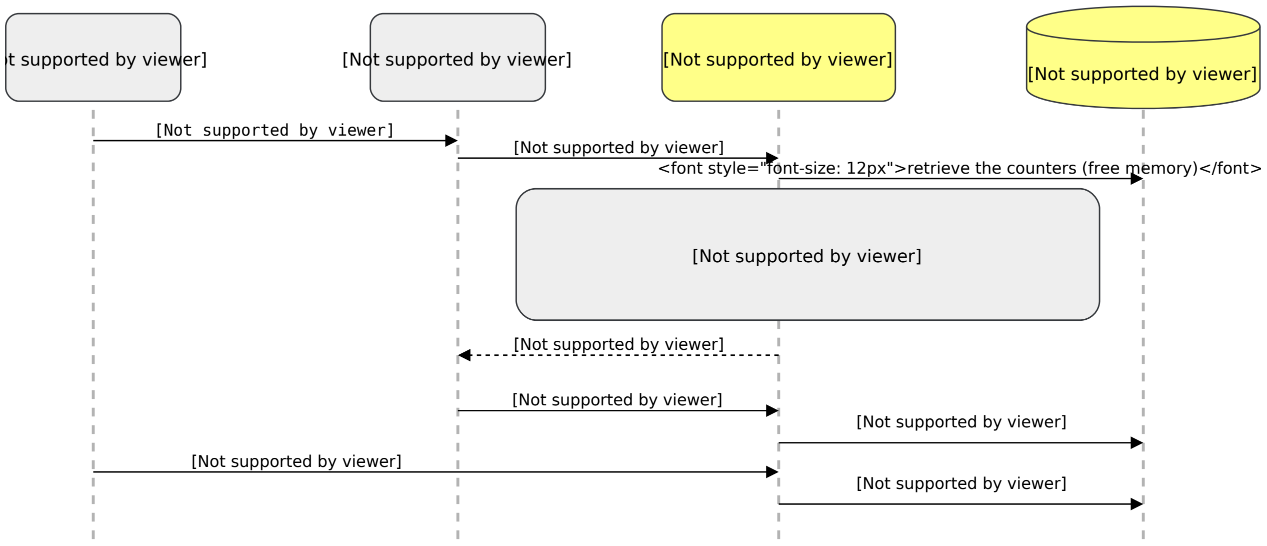
`--feature-gates=MemoryManager=true`

in order to enable the Memory Manager feature.

How Memory Manager Operates?

The Memory Manager currently offers the guaranteed memory (and hugepages) allocation for Pods in Guaranteed QoS class. To immediately put the Memory Manager into operation follow the guidelines in the section Memory Manager configuration, and subsequently, prepare and deploy a `Guaranteed` pod as illustrated in the section Placing a Pod in the Guaranteed QoS class.

The Memory Manager is a Hint Provider, and it provides topology hints for the Topology Manager which then aligns the requested resources according to these topology hints. It also enforces `cgroups` (i.e. `cpuset.mems`) for pods. The complete flow diagram concerning pod admission and deployment process is illustrated in Memory Manager KEP: Design Overview and below:


t supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
<font style="font-size: 12px">retrieve the counters (free memory)</font>
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]
[Not supported by viewer]

During this process, the Memory Manager updates its internal counters stored in Node Map and Memory Maps to manage guaranteed memory allocation.

The Memory Manager updates the Node Map during the startup and runtime as follows.

Startup

This occurs once a node administrator employs `--reserved-memory` (section Reserved memory flag). In this case, the Node Map becomes updated to reflect this reservation as illustrated in Memory Manager KEP: Memory Maps at start-up (with examples).

The administrator must provide `--reserved-memory` flag when `Static` policy is configured.

Runtime

Reference Memory Manager KEP: Memory Maps at runtime (with examples) illustrates how a successful pod deployment affects the Node Map, and it also relates to how potential Out-of-Memory (OOM) situations are handled further by Kubernetes or operating system.

Important topic in the context of Memory Manager operation is the management of NUMA groups. Each time pod's memory request is in excess of single NUMA node capacity, the Memory Manager attempts to create a group that comprises several NUMA nodes and features extend memory capacity. The problem has been solved as elaborated in Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?. Also, reference Memory Manager KEP: Simulation - how the Memory Manager works? (by examples) illustrates how the management of groups occurs.

Memory Manager configuration

Other Managers should be first pre-configured. Next, the Memory Manger feature should be enabled and be run with `Static` policy (section Static policy). Optionally, some amount of memory can be reserved for system or kubelet processes to increase node stability (section Reserved memory flag).

Policies

Memory Manager supports two policies. You can select a policy via a `kubelet` flag `--memory-manager-policy`:

- `None` (default)
- `Static`

None policy

This is the default policy and does not affect the memory allocation in any way. It acts the same as if the Memory Manager is not present at all.

The `None` policy returns default topology hint. This special hint denotes that Hint Provider (Memory Manger in this case) has no preference for NUMA affinity with any resource.

Static policy

In the case of the `Guaranteed` pod, the `Static` Memory Manger policy returns topology hints relating to the set of NUMA nodes where the memory can be guaranteed, and reserves the memory through updating the internal NodeMap object.

In the case of the `BestEffort` or `Burstable` pod, the `Static` Memory Manager policy sends back the default topology hint as there is no request for the guaranteed memory, and does not reserve the memory in the internal NodeMap object.

Reserved memory flag

The Node Allocatable mechanism is commonly used by node administrators to reserve K8S node system resources for the kubelet or operating system processes in order to enhance the node stability. A dedicated set of flags can be used for this purpose to set the total amount of reserved memory for a node. This pre-configured value is subsequently utilized to calculate the real amount of node's "allocatable" memory available to pods.

The Kubernetes scheduler incorporates "allocatable" to optimise pod scheduling process. The foregoing flags include `--kube-reserved`, `--system-reserved` and `--eviction-threshold`. The sum of their values will account for the total amount of reserved memory.

A new `--reserved-memory` flag was added to Memory Manager to allow for this total reserved memory to be split (by a node administrator) and accordingly reserved across many NUMA nodes.

The flag specifies a comma-separated list of memory reservations of different memory types per NUMA node. Memory reservations across multiple NUMA nodes can be specified using semicolon as separator. This parameter is only useful in the context of the Memory Manager feature. The Memory Manager will not use this reserved memory for the allocation of container workloads.

For example, if you have a NUMA node "NUMA0" with `10Gi` of memory available, and the `--reserved-memory` was specified to reserve `1Gi` of memory at "NUMA0", the Memory Manager assumes that only `9Gi` is available for containers.

You can omit this parameter, however, you should be aware that the quantity of reserved memory from all NUMA nodes should be equal to the quantity of memory specified by the Node Allocatable feature. If at least one node allocatable parameter is non-zero, you will need to specify `--reserved-memory` for at least one NUMA node. In fact, `eviction-hard` threshold value is equal to `100Mi` by default, so if `Static` policy is used, `--reserved-memory` is obligatory.

Also, avoid the following configurations:

1. duplicates, i.e. the same NUMA node or memory type, but with a different value;
2. setting zero limit for any of memory types;
3. NUMA node IDs that do not exist in the machine hardware;
4. memory type names different than `memory` or `hugepages-<size>` (hugepages of particular `<size>` should also exist).

Syntax:

```
--reserved-memory N:memory-type1=value1,memory-type2=value2,...
```

- `N` (integer) - NUMA node index, e.g. `0`

- `memory-type` (string) - represents memory type:
  - `memory` - conventional memory
  - `hugepages-2Mi` or `hugepages-1Gi` - hugepages
- `value` (string) - the quantity of reserved memory, e.g. `1Gi`

Example usage:

`--reserved-memory 0:memory=1Gi,hugepages-1Gi=2Gi`

or

`--reserved-memory 0:memory=1Gi --reserved-memory 1:memory=2Gi`

or

`--reserved-memory '0:memory=1Gi;1:memory=2Gi'`

When you specify values for `--reserved-memory` flag, you must comply with the setting that you prior provided via Node Allocatable Feature flags. That is, the following rule must be obeyed for each memory type:

`sum(reserved-memory(i)) = kube-reserved + system-reserved + eviction-threshold`,

where `i` is an index of a NUMA node.

If you do not follow the formula above, the Memory Manager will show an error on startup.

In other words, the example above illustrates that for the conventional memory (`type=memory`), we reserve `3Gi` in total, i.e.:

`sum(reserved-memory(i)) = reserved-memory(0) + reserved-memory(1) = 1Gi + 2Gi = 3Gi`

An example of kubelet command-line arguments relevant to the node Allocatable configuration:

- `--kube-reserved=cpu=500m,memory=50Mi`
- `--system-reserved=cpu=123m,memory=333Mi`
- `--eviction-hard=memory.available<500Mi`

Note: The default hard eviction threshold is 100MiB, and **not** zero. Remember to increase the quantity of memory that you reserve by setting `--reserved-memory` by that hard eviction threshold. Otherwise, the kubelet will not start Memory Manager and display an error.

Here is an example of a correct configuration:

```
--feature-gates=MemoryManager=true
--kube-reserved=cpu=4,memory=4Gi
--system-reserved=cpu=1,memory=1Gi
--memory-manager-policy=Static
--reserved-memory '0:memory=3Gi;1:memory=2148Mi'
```

Let us validate the configuration above:

1. `kube-reserved + system-reserved + eviction-hard(default) = reserved-memory(0) + reserved-memory(1)`
2. `4GiB + 1GiB + 100MiB = 3GiB + 2148MiB`
3. `5120MiB + 100MiB = 3072MiB + 2148MiB`
4. `5220MiB = 5220MiB` (which is correct)

Placing a Pod in the Guaranteed QoS class

If the selected policy is anything other than `None`, the Memory Manager identifies pods that are in the `Guaranteed` QoS class. The Memory Manager provides specific topology hints to the Topology Manager for each `Guaranteed` pod. For pods in a QoS class other than `Guaranteed`, the Memory Manager provides default topology hints to the Topology Manager.

The following excerpts from pod manifests assign a pod to the `Guaranteed` QoS class.

Pod with integer CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`:

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

Also, a pod sharing CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`.

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
```

Notice that both CPU and memory requests must be specified for a Pod to lend it to Guaranteed QoS class.

Troubleshooting

The following means can be used to troubleshoot the reason why a pod could not be deployed or became rejected at a node:

- pod status - indicates topology affinity errors
- system logs - include valuable information for debugging, e.g., about generated hints
- state file - the dump of internal state of the Memory Manager (includes Node Map and Memory Maps)
- starting from v1.22, the device plugin resource API can be used to retrieve information about the memory reserved for containers

Pod status (TopologyAffinityError)

This error typically occurs in the following situations:

- a node has not enough resources available to satisfy the pod's request
- the pod's request is rejected due to particular Topology Manager policy constraints

The error appears in the status of a pod:

```
kubectl get pods
```

```
NAME         READY   STATUS                  RESTARTS   AGE
guaranteed   0/1     TopologyAffinityError   0          113s
```

Use `kubectl describe pod <id>` or `kubectl get events` to obtain detailed error message:

```
Warning  TopologyAffinityError  10m   kubelet, dell8  Resources
cannot be allocated with Topology locality
```

System logs

Search system logs with respect to a particular pod.

The set of hints that Memory Manager generated for the pod can be found in the logs. Also, the set of hints generated by CPU Manager should be present in the logs.

Topology Manager merges these hints to calculate a single best hint. The best hint should be also present in the logs.

The best hint indicates where to allocate all the resources. Topology Manager tests this hint against its current policy, and based on the verdict, it either admits the pod to the node or rejects it.

Also, search the logs for occurrences associated with the Memory Manager, e.g. to find out information about `cgroups` and `cpuset.mems` updates.

Examine the memory manager state on a node

Let us first deploy a sample `Guaranteed` pod whose specification is as follows:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: guaranteed
spec:
  containers:
  - name: guaranteed
    image: consumer
    imagePullPolicy: Never
    resources:
      limits:
        cpu: "2"
        memory: 150Gi
      requests:
        cpu: "2"
        memory: 150Gi
    command: ["sleep","infinity"]
```

Next, let us log into the node where it was deployed and examine the state file in `/var/lib/kubelet/memory_manager_state`:

```json
{
   "policyName":"Static",
   "machineState":{
      "0":{
         "numberOfAssignments":1,
         "memoryMap":{
            "hugepages-1Gi":{
               "total":0,
               "systemReserved":0,
               "allocatable":0,
               "reserved":0,
               "free":0
            },
            "memory":{
               "total":134987354112,
               "systemReserved":3221225472,
               "allocatable":131766128640,
               "reserved":131766128640,
               "free":0
            }
         },
         "nodes":[
            0,
            1
         ]
      },
      "1":{
         "numberOfAssignments":1,
         "memoryMap":{
            "hugepages-1Gi":{
               "total":0,
               "systemReserved":0,
               "allocatable":0,
               "reserved":0,
               "free":0
            },
            "memory":{
               "total":135286722560,
               "systemReserved":2252341248,
               "allocatable":133034381312,
               "reserved":29295144960,
               "free":103739236352
            }
         },
         "nodes":[
            0,
            1
         ]
      }
   },
   "entries":{
      "fa9bdd38-6df9-4cf9-aa67-8c4814da37a8":{
         "guaranteed":[
            {
               "numaAffinity":[
                  0,
                  1
               ],
               "type":"memory",
               "size":161061273600
            }
         ]
      }
   },
   "checksum":4142013182
}
```

It can be deduced from the state file that the pod was pinned to both NUMA nodes, i.e.:

```
"numaAffinity":[
   0,
   1
],
```

Pinned term means that pod's memory consumption is constrained (through `cgroups` configuration) to these NUMA nodes.

This automatically implies that Memory Manager instantiated a new group that comprises these two NUMA nodes, i.e. `0` and `1` indexed NUMA nodes.

Notice that the management of groups is handled in a relatively complex manner, and further elaboration is provided in Memory Manager KEP in this and this sections.

In order to analyse memory resources available in a group,the corresponding entries from NUMA nodes belonging to the group must be added up.

For example, the total amount of free "conventional" memory in the group can be computed by adding up the free memory available at every NUMA node in the group, i.e., in the `"memory"` section of NUMA node `0` (`"free":0`) and NUMA node `1` (`"free":103739236352`). So, the total amount of free "conventional" memory in this group is equal to `0 + 103739236352` bytes.

The line `"systemReserved":3221225472` indicates that the administrator of this node reserved `3221225472` bytes (i.e. `3Gi`) to serve kubelet and system processes at NUMA node `0`, by using `--reserved-memory` flag.

Device plugin resource API

By employing the API, the information about reserved memory for each container can be retrieved, which is contained in protobuf `ContainerMemory` message. This information can be retrieved solely for pods in Guaranteed QoS class.

What's next

- Memory Manager KEP: Design Overview
- Memory Manager KEP: Memory Maps at start-up (with examples)
- Memory Manager KEP: Memory Maps at runtime (with examples)
- Memory Manager KEP: Simulation - how the Memory Manager works? (by examples)
- Memory Manager KEP: The Concept of Node Map and Memory Maps
- Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?

Configure Pods and Containers

Perform common configuration tasks for Pods and containers.

Assign Memory Resources to Containers and Pods

Assign CPU Resources to Containers and Pods

Configure GMSA for Windows Pods and containers

Configure RunAsUserName for Windows pods and containers

Create a Windows HostProcess Pod

Configure Quality of Service for Pods

Assign Extended Resources to a Container

Configure a Pod to Use a Volume for Storage

Configure a Pod to Use a PersistentVolume for Storage

Configure a Pod to Use a Projected Volume for Storage

Configure a Security Context for a Pod or Container

Configure Service Accounts for Pods

Pull an Image from a Private Registry

Configure Liveness, Readiness and Startup Probes

Assign Pods to Nodes

Assign Pods to Nodes using Node Affinity

Configure Pod Initialization

Attach Handlers to Container Lifecycle Events

Configure a Pod to Use a ConfigMap

Share Process Namespace between Containers in a Pod

Create static Pods

Translate a Docker Compose File to Kubernetes Resources

Enforce Pod Security Standards by Configuring the Built-in Admission Controller

Enforce Pod Security Standards with Namespace Labels

Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Each node in your cluster must have at least 300 MiB of memory.

A few of the steps on this page require you to run the metrics-server service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running Minikube, run the following command to enable the metrics-server:

```
minikube addons enable metrics-server
```

To see whether the metrics-server is running, or another provider of the resource metrics API (`metrics.k8s.io`), run the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output includes a reference to `metrics.k8s.io`.

```
NAME
v1beta1.metrics.k8s.io
```

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:

pods/resource/memory-request-limit.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "100Mi"
      limits:
        memory: "200Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

The `args` section in the configuration file provides arguments for the Container when it starts. The `"--vm-bytes", "150M"` arguments tell the Container to attempt to allocate 150 MiB of memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit.yaml --namespace=mem-example
```

Verify that the Pod 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:
  requests:
    memory: 100Mi
  limits:
    memory: 200Mi
...
```

Run `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod memory-demo --namespace=mem-example
```

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.

```
NAME                            CPU(cores)   MEMORY(bytes)
memory-demo                     <something>  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 can be restarted, the kubelet restarts 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 with a memory request of 50 MiB and a memory limit of 100 MiB:

pods/resource/memory-request-limit-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-2-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "50Mi"
      limits:
        memory: "100Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

In the `args` section of the configuration file, 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 apply -f https://k8s.io/examples/pods/resource/memory-
request-limit-2.yaml --namespace=mem-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

At this point, the Container might be running or killed. Repeat the preceding command until the Container is killed:

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

Get a more detailed view of the Container status:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-
example
```

The output shows that the Container was killed because it is out of memory (OOM):

```
lastState:
   terminated:
     containerID:
65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f
     exitCode: 137
     finishedAt: 2017-06-20T20:52:19Z
     reason: OOMKilled
     startedAt: null
```

The Container in this exercise can be restarted, so the kubelet restarts it. Repeat this command several times to see that the Container is repeatedly killed and restarted:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container is killed, restarted, killed again, restarted again, and so on:

```
kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          37s
```

```

kubectl get pod memory-demo-2 --namespace=mem-example
```

```
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-2   1/1       Running   2          40s
```

View detailed information about the Pod 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
66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... 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 1994 or sacrifice child
```

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 with a request for 1000 GiB of memory, which likely exceeds the capacity of any Node in your cluster.

pods/resource/memory-request-limit-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
  namespace: mem-example
spec:
  containers:
  - name: memory-demo-3-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "1000Gi"
      limits:
        memory: "1000Gi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-
request-limit-3.yaml --namespace=mem-example
```

View the Pod status:

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

The output shows that the Pod status is PENDING. That is, the Pod is not 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 predicates:: Insufficient memory (3).
```

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 do not specify a memory limit

If you do not specify a memory limit for a Container, one of the following 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 which in turn could invoke the OOM Killer. Further, in case of an OOM Kill, a container with no resource limits will have a greater chance of being killed.

- 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. Containers cannot use more CPU than the configured limit. Provided the system has CPU time free, a container is guaranteed to be allocated as much CPU as it requests.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Your cluster must have at least 1 CPU available for use to run the task examples.

A few of the steps on this page require you to run the metrics-server service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running Minikube, run the following command to enable metrics-server:

```
minikube addons enable metrics-server
```

To see whether metrics-server (or another provider of the resource metrics API, `metrics.k8s.io`) is running, type the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output will include a reference to `metrics.k8s.io`.

```
NAME
v1beta1.metrics.k8s.io
```

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 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 request of 0.5 CPU and a limit of 1 CPU. Here is the configuration file for the Pod:

pods/resource/cpu-request-limit.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
  - name: cpu-demo-ctr
    image: vish/stress
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
    args:
    - -cpus
    - "2"
```

The `args` section of the configuration file 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 apply -f https://k8s.io/examples/pods/resource/cpu-
request-limit.yaml --namespace=cpu-example
```

Verify that the Pod 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
```

Use `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod cpu-demo --namespace=cpu-example
```

This example output shows that the Pod is using 974 milliCPU, which is slightly less than the limit of 1 CPU specified in the Pod configuration.

```
NAME                        CPU(cores)   MEMORY(bytes)
cpu-demo                    974m         <something>
```

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: Another possible explanation for the CPU use being below 1.0 is that the Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require your cluster to have at least 1 CPU available for use. If your Container runs 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 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.

pods/resource/cpu-request-limit-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
  namespace: cpu-example
spec:
  containers:
  - name: cpu-demo-ctr-2
    image: vish/stress
    resources:
      limits:
        cpu: "100"
      requests:
        cpu: "100"
    args:
    - -cpus
    - "2"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-
request-limit-2.yaml --namespace=cpu-example
```

View the Pod status:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Pod 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:

```
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 predicates:: Insufficient cpu (3).
```

Delete your Pod:

```
kubectl delete pod cpu-demo-2 --namespace=cpu-example
```

If you do not specify a CPU limit

If you do not 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.

If you specify a CPU limit but do not specify a CPU request

If you specify a CPU limit for a Container but do not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

Similarly, 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.

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 Nodes. By keeping a Pod 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 GMSA for Windows Pods and containers

FEATURE STATE: `Kubernetes v1.18 [stable]`

This page shows how to configure Group Managed Service Accounts (GMSA) for Pods and containers that will run on Windows nodes. Group Managed Service Accounts are a specific type of Active Directory account that provides automatic password management, simplified service principal name (SPN) management, and the ability to delegate the management to other administrators across multiple servers.

In Kubernetes, GMSA credential specs are configured at a Kubernetes cluster-wide scope as Custom Resources. Windows Pods, as well as individual containers within a Pod, can be configured to use a GMSA for domain based functions (e.g. Kerberos authentication) when interacting with other Windows services.

Before you begin

You need to have a Kubernetes cluster and the `kubectl` command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes. This section covers a set of initial steps required once for each cluster:

Install the GMSACredentialSpec CRD

A CustomResourceDefinition(CRD) for GMSA credential spec resources needs to be configured on the cluster to define the custom resource type `GMSACredentialSpec`. Download the GMSA CRD YAML and save it as gmsa-crd.yaml. Next, install the CRD with `kubectl apply -f gmsa-crd.yaml`

Install webhooks to validate GMSA users

Two webhooks need to be configured on the Kubernetes cluster to populate and validate GMSA credential spec references at the Pod or container level:

1. A mutating webhook that expands references to GMSAs (by name from a Pod specification) into the full credential spec in JSON form within the Pod spec.

2. A validating webhook ensures all references to GMSAs are authorized to be used by the Pod service account.

Installing the above webhooks and associated objects require the steps below:

1. Create a certificate key pair (that will be used to allow the webhook container to communicate to the cluster)

2. Install a secret with the certificate from above.
3. Create a deployment for the core webhook logic.
4. Create the validating and mutating webhook configurations referring to the deployment.

A script can be used to deploy and configure the GMSA webhooks and associated objects mentioned above. The script can be run with a `--dry-run=server` option to allow you to review the changes that would be made to your cluster.

The YAML template used by the script may also be used to deploy the webhooks and associated objects manually (with appropriate substitutions for the parameters)

Configure GMSAs and Windows nodes in Active Directory

Before Pods in Kubernetes can be configured to use GMSAs, the desired GMSAs need to be provisioned in Active Directory as described in the Windows GMSA documentation. Windows worker nodes (that are part of the Kubernetes cluster) need to be configured in Active Directory to access the secret credentials associated with the desired GMSA as described in the Windows GMSA documentation

Create GMSA credential spec resources

With the GMSACredentialSpec CRD installed (as described earlier), custom resources containing GMSA credential specs can be configured. The GMSA credential spec does not contain secret or sensitive data. It is information that a container runtime can use to describe the desired GMSA of a container to Windows. GMSA credential specs can be generated in YAML format with a utility PowerShell script.

Following are the steps for generating a GMSA credential spec YAML manually in JSON format and then converting it:

1. Import the CredentialSpec module: `ipmo CredentialSpec.psm1`
2. Create a credential spec in JSON format using `New-CredentialSpec`. To create a GMSA credential spec named WebApp1, invoke `New-CredentialSpec -Name WebApp1 -AccountName WebApp1 -Domain $(Get-ADDomain -Current LocalComputer)`
3. Use `Get-CredentialSpec` to show the path of the JSON file.
4. Convert the credspec file from JSON to YAML format and apply the necessary header fields `apiVersion`, `kind`, `metadata` and `credspec` to make it a GMSACredentialSpec custom resource that can be configured in Kubernetes.

The following YAML configuration describes a GMSA credential spec named `gmsa-WebApp1`:

```yaml
apiVersion: windows.k8s.io/v1
kind: GMSACredentialSpec
metadata:
  name: gmsa-WebApp1  #This is an arbitrary name but it will be used as a reference
credspec:
  ActiveDirectoryConfig:
    GroupManagedServiceAccounts:
    - Name: WebApp1   #Username of the GMSA account
      Scope: CONTOSO  #NETBIOS Domain Name
    - Name: WebApp1   #Username of the GMSA account
      Scope: contoso.com #DNS Domain Name
  CmsPlugins:
  - ActiveDirectory
  DomainJoinConfig:
    DnsName: contoso.com  #DNS Domain Name
    DnsTreeName: contoso.com #DNS Domain Name Root
    Guid: 244818ae-87ac-4fcd-92ec-e79e5252348a  #GUID
    MachineAccountName: WebApp1 #Username of the GMSA account
    NetBiosName: CONTOSO  #NETBIOS Domain Name
    Sid: S-1-5-21-2126449477-2524075714-3094792973 #SID of GMSA
```

The above credential spec resource may be saved as `gmsa-Webapp1-credspec.yaml` and applied to the cluster using: `kubectl apply -f gmsa-Webapp1-credspec.yml`

Configure cluster role to enable RBAC on specific GMSA credential specs

A cluster role needs to be defined for each GMSA credential spec resource. This authorizes the `use` verb on a specific GMSA resource by a subject which is typically a service account. The following example shows a cluster role that authorizes usage of the `gmsa-WebApp1` credential spec from above. Save the file as gmsa-webapp1-role.yaml and apply using `kubectl apply -f gmsa-webapp1-role.yaml`

```yaml
#Create the Role to read the credspec
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: webapp1-role
rules:
- apiGroups: ["windows.k8s.io"]
  resources: ["gmsacredentialspecs"]
  verbs: ["use"]
  resourceNames: ["gmsa-WebApp1"]
```

Assign role to service accounts to use specific GMSA credspecs

A service account (that Pods will be configured with) needs to be bound to the cluster role create above. This authorizes the service account to use the desired GMSA credential spec resource. The following shows the default service account being bound to a cluster role `webapp1-role` to use `gmsa-WebApp1` credential spec resource created above.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: allow-default-svc-account-read-on-gmsa-WebApp1
  namespace: default
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole
  name: webapp1-role
  apiGroup: rbac.authorization.k8s.io
```

Configure GMSA credential spec reference in Pod spec

The Pod spec field `securityContext.windowsOptions.gmsaCredentialSpecName` is used to specify references to desired GMSA credential spec custom resources in Pod specs. This configures all containers in the Pod spec to use the specified GMSA. A sample Pod spec with the annotation populated to refer to `gmsa-WebApp1`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      securityContext:
        windowsOptions:
          gmsaCredentialSpecName: gmsa-webapp1
      containers:
      - image: mcr.microsoft.com/windows/servercore/
iis:windowsservercore-ltsc2019
        imagePullPolicy: Always
        name: iis
      nodeSelector:
        kubernetes.io/os: windows
```

Individual containers in a Pod spec can also specify the desired GMSA credspec using a per-container `securityContext.windowsOptions.gmsaCredentialSpecName` field. For example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      containers:
      - image: mcr.microsoft.com/windows/servercore/
iis:windowsservercore-ltsc2019
        imagePullPolicy: Always
        name: iis
        securityContext:
          windowsOptions:
            gmsaCredentialSpecName: gmsa-Webapp1
      nodeSelector:
        kubernetes.io/os: windows
```

As Pod specs with GMSA fields populated (as described above) are applied in a cluster, the following sequence of events take place:

1. The mutating webhook resolves and expands all references to GMSA credential spec resources to the contents of the GMSA credential spec.

2. The validating webhook ensures the service account associated with the Pod is authorized for the `use` verb on the specified GMSA credential spec.

3. The container runtime configures each Windows container with the specified GMSA credential spec so that the container can assume the identity of the GMSA in Active Directory and access services in the domain using that identity.

Authenticating to network shares using hostname or FQDN

If you are experiencing issues connecting to SMB shares from Pods using hostname or FQDN, but are able to access the shares via their IPv4 address then make sure the following registry key is set on the Windows nodes.

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\hns\State" /v
EnableCompartmentNamespace /t REG_DWORD /d 1
```

Running Pods will then need to be recreated to pick up the behavior changes. More information on how this registry key is used can be found here

Troubleshooting

If you are having difficulties getting GMSA to work in your environment, there are a few troubleshooting steps you can take.

First, make sure the credspec has been passed to the Pod. To do this you will need to `exec` into one of your Pods and check the output of the `nltest.exe /parentdomain` command.

In the example below the Pod did not get the credspec correctly:

```
kubectl exec -it iis-auth-7776966999-n5nzr powershell.exe
```

`nltest.exe /parentdomain` results in the following error:

```
Getting parent domain failed: Status = 1722 0x6ba
RPC_S_SERVER_UNAVAILABLE
```

If your Pod did get the credspec correctly, then next check communication with the domain. First, from inside of your Pod, quickly do an nslookup to find the root of your domain.

This will tell us 3 things:

1. The Pod can reach the DC
2. The DC can reach the Pod
3. DNS is working correctly.

If the DNS and communication test passes, next you will need to check if the Pod has established secure channel communication with the domain. To do this, again, `exec` into your Pod and run the `nltest.exe /query` command.

```
nltest.exe /query
```

Results in the following output:

```
I_NetLogonControl failed: Status = 1722 0x6ba
RPC_S_SERVER_UNAVAILABLE
```

This tells us that for some reason, the Pod was unable to logon to the domain using the account specified in the credspec. You can try to repair the secure channel by running the following:

```
nltest /sc_reset:domain.example
```

If the command is successful you will see and output similar to this:

```
Flags: 30 HAS_IP  HAS_TIMESERV
Trusted DC Name \\dc10.domain.example
Trusted DC Connection Status Status = 0 0x0 NERR_Success
The command completed successfully
```

If the above corrects the error, you can automate the step by adding the following lifecycle hook to your Pod spec. If it did not correct the error, you will need to examine your credspec again and confirm that it is correct and complete.

```
        image: registry.domain.example/iis-auth:1809v1
        lifecycle:
          postStart:
            exec:
              command: ["powershell.exe","-command","do
{ Restart-Service -Name netlogon } while ( $($Result =
(nltest.exe /query); if ($Result -like '*0x0 NERR_Success*')
{return $true} else {return $false}) -eq $false)"]
        imagePullPolicy: IfNotPresent
```

If you add the `lifecycle` section show above to your Pod spec, the Pod will execute the commands listed to restart the `netlogon` service until the `nltest.exe /query` command exits without error.

Configure RunAsUserName for Windows pods and containers

FEATURE STATE: `Kubernetes v1.18 [stable]`

This page shows how to use the `runAsUserName` setting for Pods and containers that will run on Windows nodes. This is roughly equivalent of the Linux-specific `runAsUser` setting, allowing you to run applications in a container as a different username than the default.

Before you begin

You need to have a Kubernetes cluster and the kubectl command-line tool must be configured to communicate with your cluster. The cluster is expected to have Windows worker nodes where pods with containers running Windows workloads will get scheduled.

Set the Username for a Pod

To specify the username with which to execute the Pod's container processes, include the `securityContext` field (PodSecurityContext) in the Pod specification, and within it, the `windowsOptions` (WindowsSecurityContextOptions) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Pod apply to all Containers and init Containers in the Pod.

Here is a configuration file for a Windows Pod that has the `runAsUserName` field set:

windows/run-as-username-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
  nodeSelector:
    kubernetes.io/os: windows
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-pod.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-pod-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-pod-demo -- powershell
```

Check that the shell is running user the correct username:

```
echo $env:USERNAME
```

The output should be:

```
ContainerUser
```

Set the Username for a Container

To specify the username with which to execute a Container's processes, include the `securityContext` field (SecurityContext) in the Container manifest, and within it, the `windowsOptions` (WindowsSecurityContextOptions) field containing the `runAsUserName` field.

The Windows security context options that you specify for a Container apply only to that individual Container, and they override the settings made at the Pod level.

Here is the configuration file for a Pod that has one Container, and the `runAsUserName` field is set at the Pod level and the Container level:

windows/run-as-username-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-container.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod run-as-username-container-demo
```

Get a shell to the running Container:

```
kubectl exec -it run-as-username-container-demo -- powershell
```

Check that the shell is running user the correct username (the one set at the Container level):

```
echo $env:USERNAME
```

The output should be:

```
ContainerAdministrator
```

Windows Username limitations

In order to use this feature, the value set in the `runAsUserName` field must be a valid username. It must have the following format: `DOMAIN\USER`, where `DOMAIN\` is optional. Windows user names are case insensitive. Additionally, there are some restrictions regarding the `DOMAIN` and `USER`:

- The `runAsUserName` field cannot be empty, and it cannot contain control characters (ASCII values: `0x00-0x1F`, `0x7F`)
- The `DOMAIN` must be either a NetBios name, or a DNS name, each with their own restrictions:
  - NetBios names: maximum 15 characters, cannot start with `.` (dot), and cannot contain the following characters: `\ / : * ? " < > |`
  - DNS names: maximum 255 characters, contains only alphanumeric characters, dots, and dashes, and it cannot start or end with a `.` (dot) or `-` (dash).
- The `USER` must have at most 20 characters, it cannot contain *only* dots or spaces, and it cannot contain the following characters: `" / \ [ ] : ; | = , + * ? < > @`.

Examples of acceptable values for the `runAsUserName` field: `ContainerAdministrator`, `ContainerUser`, `NT AUTHORITY\NETWORK SERVICE`, `NT AUTHORITY\LOCAL SERVICE`.

For more information about these limtations, check here and here.

What's next

- Guide for scheduling Windows containers in Kubernetes
- Managing Workload Identity with Group Managed Service Accounts (GMSA)
- Configure GMSA for Windows pods and containers

Create a Windows HostProcess Pod

FEATURE STATE: `Kubernetes v1.23 [beta]`

Windows HostProcess containers enable you to run containerized workloads on a Windows host. These containers operate as normal processes but have access to the host network namespace, storage, and devices when given the appropriate user privileges. HostProcess containers can be used to deploy network plugins, storage configurations, device plugins, kube-proxy, and other components to Windows nodes without the need for dedicated proxies or the direct installation of host services.

Administrative tasks such as installation of security patches, event log collection, and more can be performed without requiring cluster operators to log onto each Windows node. HostProcess containers can run as any user that is available on the host or is in the domain of the host machine, allowing administrators to restrict resource access through user permissions. While neither filesystem or process isolation are supported, a new volume is created on the host upon starting the container to give it a clean and consolidated workspace. HostProcess containers can also be built on top of existing Windows base images and do not inherit the same compatibility requirements as Windows server containers, meaning that the version of the base images does not need to match that of the host. It is, however, recommended that you use the same base image version as your Windows Server container workloads to ensure you do not have any unused images taking up space on the node. HostProcess containers also support volume mounts within the container volume.

When should I use a Windows HostProcess container?

- When you need to perform tasks which require the networking namespace of the host. HostProcess containers have access to the host's network interfaces and IP addresses.
- You need access to resources on the host such as the filesystem, event logs, etc.
- Installation of specific device drivers or Windows services.
- Consolidation of administrative tasks and security policies. This reduces the degree of privileges needed by Windows nodes.

Before you begin

This task guide is specific to Kubernetes v1.23. If you are not running Kubernetes v1.23, check the documentation for that version of Kubernetes.

In Kubernetes 1.23, the HostProcess container feature is enabled by default. The kubelet will communicate with containerd directly by passing the hostprocess flag via CRI. You can use the latest version of containerd (v1.6+) to run HostProcess containers. How to install containerd.

To *disable* HostProcess containers you need to pass the following feature gate flag to the **kubelet** and **kube-apiserver**:

```
--feature-gates=WindowsHostProcessContainers=false
```

See Features Gates documentation for more details.

Limitations

These limitations are relevant for Kubernetes v1.23:

- HostProcess containers require containerd 1.6 or higher container runtime.
- HostProcess pods can only contain HostProcess containers. This is a current limitation of the Windows OS; non-privileged Windows containers cannot share a vNIC with the host IP namespace.
- HostProcess containers run as a process on the host and do not have any degree of isolation other than resource constraints imposed on the HostProcess user account. Neither filesystem or Hyper-V isolation are supported for HostProcess containers.
- Volume mounts are supported and are mounted under the container volume. See Volume Mounts
- A limited set of host user accounts are available for HostProcess containers by default. See Choosing a User Account.
- Resource limits (disk, memory, cpu count) are supported in the same fashion as processes on the host.
- Both Named pipe mounts and Unix domain sockets are **not** supported and should instead be accessed via their path on the host (e.g. \\.\pipe*)

HostProcess Pod configuration requirements

Enabling a Windows HostProcess pod requires setting the right configurations in the pod security configuration. Of the policies defined in the Pod Security Standards HostProcess pods are disallowed by the baseline and restricted policies. It is therefore recommended that HostProcess pods run in alignment with the privileged profile.

When running under the privileged policy, here are the configurations which need to be set to enable the creation of a HostProcess pod:

Control	**Policy**
`securityContext.windowsOptions.hostProcess`	Windows pods offer the ability to run HostProcess containers which enables privileged access to the Windows node. **Allowed Values** • `true`

Control	Policy
hostNetwork	Will be in host network by default initially. Support to set network to a different compartment may be desirable in the future. **Allowed Values** • `true`
securityContext.windowsOptions.runAsUsername	Specification of which user the HostProcess container should run as is required for the pod spec. **Allowed Values** • `NT AUTHORITY\SYSTEM` • `NT AUTHORITY\Local service` • `NT AUTHORITY\NetworkService`
runAsNonRoot	Because HostProcess containers have privileged access to the host, the `runAsNonRoot` field cannot be set to true. **Allowed Values** • Undefined/Nil • `false`

Example manifest (excerpt)

```yaml
spec:
  securityContext:
    windowsOptions:
      hostProcess: true
      runAsUserName: "NT AUTHORITY\\Local service"
  hostNetwork: true
  containers:
  - name: test
    image: image1:latest
    command:
      - ping
      - -t
      - 127.0.0.1
  nodeSelector:
    "kubernetes.io/os": windows
```

Volume mounts

HostProcess containers support the ability to mount volumes within the container volume space. Applications running inside the container can access volume mounts directly via relative or absolute paths. An environment variable `$CONTAINER_SANDBOX_MOUNT_POINT` is set upon container creation and provides the absolute host path to the container volume. Relative paths are based upon the `.spec.containers.volumeMounts.mountPath` configuration.

Example

To access service account tokens the following path structures are supported within the container:

`.\var\run\secrets\kubernetes.io\serviceaccount\`

`$CONTAINER_SANDBOX_MOUNT_POINT\var\run\secrets\kubernetes.io\serviceaccount\`

Resource limits

Resource limits (disk, memory, cpu count) are applied to the job and are job wide. For example, with a limit of 10MB set, the memory allocated for any HostProcess job object will be capped at 10MB. This is the same behavior as other Windows container types. These limits would be specified the same way they are currently for whatever orchestrator or runtime is being used. The only difference is in the disk resource usage calculation used for resource tracking due to the difference in how HostProcess containers are bootstrapped.

Choosing a user account

HostProcess containers support the ability to run as one of three supported Windows service accounts:

- **LocalSystem**
- **LocalService**
- **NetworkService**

You should select an appropriate Windows service account for each HostProcess container, aiming to limit the degree of privileges so as to avoid accidental (or even malicious) damage to the host. The LocalSystem service account has the highest level of privilege of the three and should be used only if absolutely necessary. Where possible, use the LocalService service account as it is the least privileged of the three options.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.
- For every Container in the Pod, the memory limit must equal the memory request.
- Every Container in the Pod must have a CPU limit and a CPU request.

- For every Container in the Pod, the CPU limit must equal the CPU request.

These restrictions apply to init containers and app containers equally.

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:

pods/qos/qos-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod.yaml --
namespace=qos-example
```

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 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
    ...
status:
  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 or limit.

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.

pods/qos/qos-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-2
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-2.yaml
--namespace=qos-example
```

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
  ...
status:
  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:

pods/qos/qos-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-3
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-3-ctr
    image: nginx
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-3.yaml
--namespace=qos-example
```

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: {}
  ...
status:
  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.

pods/qos/qos-pod-4.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-4
  namespace: qos-example
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 apply -f https://k8s.io/examples/pods/qos/qos-pod-4.yaml
--namespace=qos-example
```

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:

```yaml
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
    ...
status:
  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
- Control Topology Management policies on a node

Assign Extended Resources to a Container

FEATURE STATE: `Kubernetes v1.23 [stable]`

This page shows how to assign extended resources to a 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Before you do this exercise, do the exercise in Advertise Extended Resources for a Node. That will configure one of your Nodes to advertise a dongle resource.

Assign an extended resource to a Pod

To request an extended resource, include the `resources:requests` field in your Container manifest. Extended resources are fully qualified with any domain outside of `*.kubernetes.io/`. Valid extended resource names have the form `example.com/foo` where `example.com` is replaced with your organization's domain and `foo` is a descriptive resource name.

Here is the configuration file for a Pod that has one Container:

`pods/resource/extended-resource-pod.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo
spec:
  containers:
  - name: extended-resource-demo-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 3
      limits:
        example.com/dongle: 3
```

In the configuration file, you can see that the Container requests 3 dongles.

Create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/extended-resource-pod.yaml
```

Verify that the Pod is running:

```
kubectl get pod extended-resource-demo
```

Describe the Pod:

```
kubectl describe pod extended-resource-demo
```

The output shows dongle requests:

```
Limits:
  example.com/dongle: 3
Requests:
  example.com/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.

pods/resource/extended-resource-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo-2
spec:
  containers:
  - name: extended-resource-demo-2-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 2
      limits:
        example.com/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 apply -f https://k8s.io/examples/pods/resource/extended-
resource-pod-2.yaml
```

Describe the Pod

```
kubectl describe pod extended-resource-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 (extended-resource-demo-2)
failed to fit in any node
fit failure summary on nodes : Insufficient example.com/dongle
(1)
```

View the Pod status:

```
kubectl get pod extended-resource-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
extended-resource-demo-2    0/1       Pending   0          6m
```

Clean up

Delete the Pods that you created for this exercise:

```
kubectl delete pod extended-resource-demo
kubectl delete pod extended-resource-demo-2
```

What's next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Advertise Extended 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, filesystem changes 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 (such as Redis) and databases.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

`pods/storage/redis.yaml`

```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 apply -f https://k8s.io/examples/pods/storage/
redis.yaml
```

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 then create a file:

   ```
   root@redis:/data# cd /data/redis/
   root@redis:/data/redis# echo Hello > test-file
   ```

5. In your shell, list the running processes:

   ```
   root@redis:/data/redis# apt-get update
   root@redis:/data/redis# apt-get install procps
   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 *:6379
   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, go to `/data/redis`, and verify that `test-file` is still there.

   ```
   root@redis:/data/redis# cd /data/redis/
   root@redis:/data/redis# ls
   test-file
   ```

3. Delete the Pod that you created for this exercise:

   ```
   kubectl delete pod redis
   ```

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 you how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. You, as cluster administrator, create a PersistentVolume backed by physical storage. You do not associate the volume with any Pod.
2. You, now taking the role of a developer / cluster user, create a PersistentVolumeClaim that is automatically bound to a suitable PersistentVolume.
3. You create a Pod that uses the above PersistentVolumeClaim for 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 single 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 on that Node, create a `/mnt/data` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data
```

In the `/mnt/data` directory, create an `index.html` file:

```
# This again assumes that your Node uses "sudo" to run commands
# as the superuser
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/
index.html"
```

Note: If your Node uses a tool for superuser access other than `sudo`, you can usually make this work if you replace `sudo` with the name of the other tool.

Test that the `index.html` file exists:

```
cat /mnt/data/index.html
```

The output should be:

```
Hello from Kubernetes storage
```

You can now close the shell to your Node.

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:

pods/storage/pv-volume.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

The configuration file specifies that the volume is at `/mnt/data` on 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 apply -f https://k8s.io/examples/pods/storage/pv-
volume.yaml
```

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     STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO           Retain
Available             manual                   4s
```

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:

pods/storage/pv-claim.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-
claim.yaml
```

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                   STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO           Retain
Bound     default/task-pv-claim   manual                   2m
```

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   AGE
task-pv-claim   Bound     task-pv-volume   10Gi
RWO           manual           30s
```

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:

pods/storage/pv-pod.yaml

```
apiVersion: v1
kind: Pod
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 apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

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:

```
# Be sure to run these 3 commands inside the root shell that
comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

If you see that message, you have successfully configured a Pod to use storage from a PersistentVolumeClaim.

Clean up

Delete the Pod, the PersistentVolumeClaim and the PersistentVolume:

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

If you don't already have a shell open to the Node in your cluster, open a new shell the same way that you did earlier.

In the shell on your Node, remove the file and directory that you created:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo rm /mnt/data/index.html
sudo rmdir /mnt/data
```

You can now close the shell to your Node.

Mounting the same persistentVolume in two places

pods/storage/pv-duplicate.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: test
      image: nginx
      volumeMounts:
        # a mount for site-data
        - name: config
          mountPath: /usr/share/nginx/html
          subPath: html
        # another mount for nginx config
        - name: config
          mountPath: /etc/nginx/nginx.conf
        subPath: nginx.conf
  volumes:
    - name: config
      persistentVolumeClaim:
        claimName: test-nfs-claim
```

You can perform 2 volume mounts on your nginx container:

`/usr/share/nginx/html` for the static website `/etc/nginx/nginx.conf` for the default config

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:

```
apiVersion: v1
kind: PersistentVolume
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`, `downwardAPI`, and `serviceAccountToken` volumes can be projected.

Note: `serviceAccountToken` is not a volume type.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

pods/storage/projected.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox:1.28
    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 apply -f https://k8s.io/examples/pods/storage/
   projected.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/
   ```

Clean up

Delete the Pod and the Secrets:

```
kubectl delete pod test-projected-volume
kubectl delete secret user pass
```

What's next

- Learn more about projected volumes.

- Read 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, but are not limited to:

- 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: Filter a process's system calls.
- `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 always true when the container:
  - is run as privileged, or
  - has `CAP_SYS_ADMIN`
- `readOnlyRootFilesystem`: Mounts the container's root filesystem as read-only.

The above bullets are not a complete set of security context settings -- please see SecurityContext for a comprehensive list.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

pods/security/security-context.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox:1.28
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, all processes run with user ID 1000. The `runAsGroup` field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when `runAsGroup` is specified. Since `fsGroup` field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume `/data/demo` and any files created in that volume will be Group ID 2000.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context.yaml
```

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
```

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

```
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 1h
    6 1000      0:00 sh
...
```

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
```

Run the following command:

```
id
```

The output is similar to this:

```
uid=1000 gid=3000 groups=2000
```

From the output, you can see that `gid` is 3000 which is same as the `runAsGroup` field. If the `runAsGroup` was omitted, the `gid` would remain as 0 (root) and the process will be able to interact with files that are owned by the root(0) group and groups that have the required group permissions for the root (0) group.

Exit your shell:

```
exit
```

Configure volume permission and ownership change policy for Pods

FEATURE STATE: `Kubernetes v1.23 [stable]`

By default, Kubernetes recursively changes ownership and permissions for the contents of each volume to match the `fsGroup` specified in a Pod's `securityContext` when that volume is mounted. For large volumes, checking and changing ownership and permissions can take a lot of time, slowing Pod startup. You can use the `fsGroupChangePolicy` field inside a `securityContext` to control the way that Kubernetes checks and manages ownership and permissions for a volume.

fsGroupChangePolicy - `fsGroupChangePolicy` defines behavior for changing ownership and permission of the volume before being exposed inside a Pod. This field only applies to volume types that support `fsGroup` controlled ownership and permissions. This field has two possible values:

- *OnRootMismatch*: Only change permissions and ownership if permission and ownership of root directory does not match with expected permissions of the volume. This could help shorten the time it takes to change ownership and permission of a volume.
- *Always*: Always change permission and ownership of the volume when volume is mounted.

For example:

```yaml
securityContext:
  runAsUser: 1000
  runAsGroup: 3000
  fsGroup: 2000
  fsGroupChangePolicy: "OnRootMismatch"
```

Note: This field has no effect on ephemeral volume types such as `secret`, `configMap`, and `emptydir`.

Delegating volume permission and ownership change to CSI driver

FEATURE STATE: `Kubernetes v1.23 [beta]`

If you deploy a Container Storage Interface (CSI) driver which supports the `VOLUME_MOUNT_GROUP` `NodeServiceCapability`, the process of setting file ownership and permissions based on the `fsGroup` specified in the `securityContext` will be performed by the CSI driver instead of Kubernetes, provided that the `DelegateFSGroupToCSIDriver` Kubernetes feature gate is enabled. In this case, since Kubernetes doesn't perform any ownership and permission change, `fsGroupChangePolicy` does not take effect, and as specified by CSI, the driver is expected to mount the volume with the provided `fsGroup`, resulting in a volume that is readable/writable by the `fsGroup`.

Please refer to the KEP and the description of the `VolumeCapability.MountVolume.volume_mount_group` field in the CSI spec for more information.

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:

pods/security/security-context-2.yaml

```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 apply -f https://k8s.io/examples/pods/security/security-context-2.yaml
```

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 server.js
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:

pods/security/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 apply -f https://k8s.io/examples/pods/security/security-
context-3.yaml
```

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:

pods/security/security-context-4.yaml

```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 apply -f https://k8s.io/examples/pods/security/security-
context-4.yaml
```

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.

Set the Seccomp Profile for a Container

To set the Seccomp profile for a Container, include the `seccompProfile` field in the `securityContext` section of your Pod or Container manifest. The `seccompProfile` field is a SeccompProfile object consisting of `type` and `localhostProfile`. Valid options for `type` include `RuntimeDefault`, `Unconfined`, and `Localhost`. `localhostProfile` must only be set set if `type: Localhost`. It indicates the path of the pre-configured profile on the node, relative to the kubelet's configured Seccomp profile location (configured with the `--root-dir` flag).

Here is an example that sets the Seccomp profile to the node's container runtime default profile:

```
...
securityContext:
  seccompProfile:
    type: RuntimeDefault
```

Here is an example that sets the Seccomp profile to a pre-configured file at `<kubelet-root-dir>/seccomp/my-profiles/profile-allow.json`:

```
...
securityContext:
  seccompProfile:
    type: Localhost
    localhostProfile: my-profiles/profile-allow.json
```

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.

Clean up

Delete the Pod:

```
kubectl delete pod security-context-demo
kubectl delete pod security-context-demo-2
kubectl delete pod security-context-demo-3
kubectl delete pod security-context-demo-4
```

What's next

- [PodSecurityContext](#)

- SecurityContext
- Tuning Docker with the newest security enhancements
- Security Contexts design document
- Ownership Management design document
- PodSecurityPolicy
- AllowPrivilegeEscalation design document
- For more information about security mechanisms in Linux, see Overview of Linux Kernel Security Features (Note: Some information is out of date)

Configure Service Accounts for Pods

A service account provides an identity for processes that run in a Pod.

Note: This document is a user introduction to Service Accounts and 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 (for example, 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 (for example, `default`).

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 (for example, `kub` `ectl 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 of the service account 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
```

The output is similar to this:

```
NAME      SECRETS    AGE
default   1          1d
```

You can create additional ServiceAccount objects like this:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
```

The name of a ServiceAccount object must be a valid DNS subdomain name.

If you get a complete dump of the service account object, like this:

```
kubectl get serviceaccounts/build-robot -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  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, 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.

```
kubectl apply -f - <<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
```

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
```

The output is similar to this:

```
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-002dc52800da

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

Create an imagePullSecret

- Create an imagePullSecret, as described in Specifying ImagePullSecrets on a Pod.

  ```
  kubectl create secret docker-registry myregistrykey --docker-
  server=DUMMY_SERVER \
          --docker-username=DUMMY_USERNAME --docker-password=DU
  MMY_DOCKER_PASSWORD \
          --docker-email=DUMMY_DOCKER_EMAIL
  ```

- Verify it has been created.

  ```
  kubectl get secrets myregistrykey
  ```

  The output is similar to this:

  ```
  NAME                 TYPE                              DATA
  AGE
  myregistrykey Â   kubernetes.io/.dockerconfigjson Â  1 Â  Â
  Â  1d
  ```

Add image pull secret to service account

Next, modify the default service account for the namespace to use this secret as an imagePullSecret.

```
kubectl patch serviceaccount default -p '{"imagePullSecrets":
[{"name": "myregistrykey"}]}'
```

You can instead use `kubectl edit`, or manually edit the YAML manifests as shown below:

```
kubectl get serviceaccounts default -o yaml > ./sa.yaml
```

The output of the `sa.yaml` file is similar to this:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
```

Using your editor of choice (for example `vi`), open the `sa.yaml` file, delete line with key `resourceVersion`, add lines with `imagePullSecrets:` and save.

The output of the `sa.yaml` file is similar to this:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
```

Finally replace the serviceaccount with the new updated `sa.yaml` file

```
kubectl replace serviceaccount default -f ./sa.yaml
```

Verify imagePullSecrets was added to pod spec

Now, when a new Pod is created in the current namespace and using the default ServiceAccount, the new Pod has its `spec.imagePullSecrets` field set automatically:

```
kubectl run nginx --image=nginx --restart=Never
kubectl get pod nginx -o=jsonpath='{.spec.imagePullSecrets[0].nam
e}{"\n"}'
```

The output is:

```
myregistrykey
```

Service Account Token Volume Projection

FEATURE STATE: `Kubernetes v1.20 [stable]`

Note:

To enable and use token request projection, you must specify each of the following command line arguments to `kube-apiserver`:

- `--service-account-issuer`

  It can be used as the Identifier of the service account token issuer. You can specify the `--service-account-issuer` argument multiple times, this can be useful to enable a non-disruptive change of the issuer. When this flag is specified multiple times, the first is used to generate tokens and all are used to determine which issuers are accepted. You must be running Kubernetes v1.22 or later to be able to specify `--service-account-issuer` multiple times.

- `--service-account-key-file`

  File containing PEM-encoded x509 RSA or ECDSA private or public keys, used to verify ServiceAccount tokens. The specified file can contain multiple keys, and the flag can be specified multiple times with different files. If specified multiple times, tokens signed by any of the specified keys are considered valid by the Kubernetes API server.

- `--service-account-signing-key-file`

  Path to the file that contains the current private key of the service account token issuer. The issuer signs issued ID tokens with this private key.

- `--api-audiences` (can be omitted)

  The service account token authenticator validates that tokens used against the API are bound to at least one of these audiences. If `api-audiences` is specified multiple times, tokens for any of the specified audiences are considered valid by the Kubernetes API server. If the `--service-account-issuer` flag is configured and this flag is not, this field defaults to a single element list containing the issuer URL.

The kubelet can also project a service account token into a Pod. You can specify desired properties of the token, such as the audience and the validity duration. These properties are not configurable on the default service account token. The service account token will also become invalid against the API when the Pod or the ServiceAccount is deleted.

This behavior is configured on a PodSpec using a ProjectedVolume type called ServiceAccountToken. To provide a pod with a token with an audience of "vault" and a validity duration of two hours, you would configure the following in your PodSpec:

pods/pod-projected-svc-token.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: vault-token
  serviceAccountName: build-robot
  volumes:
  - name: vault-token
    projected:
      sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/pod-projected-svc-token.yaml
```

The kubelet will request and store the token on behalf of the pod, make the token available to the pod at a configurable file path, and refresh the token as it approaches expiration. The kubelet proactively rotates the token if it is older than 80% of its total TTL, or if the token is older than 24 hours.

The application is responsible for reloading the token when it rotates. Periodic reloading (e.g. once every 5 minutes) is sufficient for most use cases.

Service Account Issuer Discovery

FEATURE STATE: `Kubernetes v1.21 [stable]`

The Service Account Issuer Discovery feature is enabled when the Service Account Token Projection feature is enabled, as described above.

Note:

The issuer URL must comply with the OIDC Discovery Spec. In practice, this means it must use the `https` scheme, and should serve an OpenID provider configuration at `{service-account-issuer}/.well-known/openid-configuration`.

If the URL does not comply, the `ServiceAccountIssuerDiscovery` endpoints will not be registered, even if the feature is enabled.

The Service Account Issuer Discovery feature enables federation of Kubernetes service account tokens issued by a cluster (the *identity provider*) with external systems (*relying parties*).

When enabled, the Kubernetes API server provides an OpenID Provider Configuration document at `/.well-known/openid-configuration` and the associated JSON Web Key Set (JWKS) at `/openid/v1/jwks`. The OpenID Provider Configuration is sometimes referred to as the *discovery document*.

Clusters include a default RBAC ClusterRole called `system:service-account-issuer-discovery`. A default RBAC ClusterRoleBinding assigns this role to the `system:serviceaccounts` group, which all service accounts implicitly belong to. This allows pods running on the cluster to access the service account discovery document via their mounted service account token. Administrators may, additionally, choose to bind the role to `system:authenticated` or `system:unauthenticated` depending on their security requirements and which external systems they intend to federate with.

Note: The responses served at `/.well-known/openid-configuration` and `/openid/v1/jwks` are designed to be OIDC compatible, but not strictly OIDC compliant. Those documents contain only the parameters necessary to perform validation of Kubernetes service account tokens.

The JWKS response contains public keys that a relying party can use to validate the Kubernetes service account tokens. Relying parties first query for the OpenID Provider Configuration, and use the `jwks_uri` field in the response to find the JWKS.

In many cases, Kubernetes API servers are not available on the public internet, but public endpoints that serve cached responses from the API server can be made available by users or service providers. In these cases, it is possible to override the `jwks_uri` in the OpenID Provider Configuration so that it points to the public endpoint, rather than the API server's address, by passing the `--service-account-jwks-uri` flag to the API server. Like the issuer URL, the JWKS URI is required to use the `https` scheme.

What's next

See also:

- Cluster Admin Guide to Service Accounts
- Service Account Signing Key Retrieval KEP

- OIDC Discovery Spec

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 container image registry or repository. There are many private registries in use. This task uses Docker Hub as an example registry.

ðŸ›‡ This item links to a third party project or product that is not part of Kubernetes itself. More information

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 the `docker` command line tool, and a Docker ID for which you know the password.
- If you are using a different private container registry, you need the command line tool for that registry and any login information for the registry.

Log in to Docker Hub

On your laptop, you must authenticate with a registry in order to pull a private image.

Use the `docker` tool to log in to Docker Hub. See the *log in* section of Docker ID accounts for more information.

```
docker login
```

When prompted, enter your Docker ID, and then the credential you want to use (access token, or the password for your Docker ID).

The login process creates or updates a `config.json` file that holds an authorization token. Review how Kubernetes interprets this file.

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 based on existing credentials

A Kubernetes cluster uses the Secret of `kubernetes.io/dockerconfigjson` type to authenticate with a container registry to pull a private image.

If you already ran `docker login`, you can copy that credential into Kubernetes:

```
kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

If you need more control (for example, to set a namespace or a label on the new secret) then you can customise the Secret before storing it. Be sure to:

- set the name of the data item to `.dockerconfigjson`
- base64 encode the Docker configuration file and then 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: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFhYWFhYWFh
YWFhYWFhYWFhYWFhYWFhYWFhYWxsbGxsbGxsbGxsbGxsbGxsbGxsbGxsbGxsb
Gx5eXl5eXl5eXl5eXl5eXl5eSBsbGxsbGxsbGxsbGxsbG9vb29vb29vb29vb2
9vb29vb29vb29vb29vb25ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2d
nZ2dnZ2dnZ2cgYXV0aCBrZXlzCg==
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 `S` `ecret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value ...`, it means the base64 encoded string in the data was successfully decoded, but could not be parsed as a `.docker/config.json` file.

Create a Secret by providing credentials on the command line

Create this Secret, naming it `regcred`:

```
kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

where:

- `<your-registry-server>` is your Private Docker Registry FQDN. Use `https://index.docker.io/v1/` for DockerHub.
- `<your-name>` is your Docker username.
- `<your-pword>` is your Docker password.
- `<your-email>` is your Docker email.

You have successfully set your Docker credentials in the cluster as a Secret called `regcred`.

Note: Typing secrets on the command line may store them in your shell history unprotected, and those secrets might also be visible to other users on your PC during the time that `kubectl` is running.

Inspecting the Secret `regcred`

To understand the contents of the `regcred` Secret you created, start by viewing the Secret in YAML format:

```
kubectl get secret regcred --output=yaml
```

The output is similar to this:

```yaml
apiVersion: v1
kind: Secret
metadata:
  ...
  name: regcred
  ...
data:
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
type: kubernetes.io/dockerconfigjson
```

The value of the `.dockerconfigjson` field is a base64 representation of your Docker credentials.

To understand what is in the `.dockerconfigjson` field, convert the secret data to a readable format:

```
kubectl get secret regcred --output="jsonpath={.data.
\.dockerconfigjson}" | base64 --decode
```

The output is similar to this:

```
{"auths":{"your.private.registry.example.com":{"username":"janedo
e","password":"xxxxxxxxxxx","email":"jdoe@example.com","auth":"c3
R...zE2"}}}
```

To understand what is in the `auth` field, convert the base64-encoded data to a readable format:

```
echo "c3R...zE2" | base64 --decode
```

The output, username and password concatenated with a `:`, is similar to this:

```
janedoe:xxxxxxxxxxx
```

Notice that the Secret data contains the authorization token similar to your local `~/.docker/config.json` file.

You have successfully set your Docker credentials as a Secret called `regcred` in the cluster.

Create a Pod that uses your Secret

Here is a manifest for an example Pod that needs access to your Docker credentials in `regcred`:

pods/private-reg-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcred
```

Download the above file onto your computer:

```
curl -L -O my-private-reg-pod.yaml https://k8s.io/examples/pods/
private-reg-pod.yaml
```

In file `my-private-reg-pod.yaml`, replace `<your-private-image>` with the path to an image in a private registry such as:

```
your.private.registry.example.com/janedoe/jdoe-private:v1
```

To pull the image from the private registry, Kubernetes needs credentials. The `imagePullSecrets` field in the configuration file specifies that Kubernetes should get the credentials from a Secret named `regcred`.

Create a Pod that uses your Secret, and verify that the Pod is running:

```
kubectl apply -f my-private-reg-pod.yaml
kubectl get pod private-reg
```

What's next

- Learn more about Secrets
  - or read the API reference for Secret
- Learn more about using a private registry.
- Learn more about adding image pull secrets to a service account.
- See kubectl create secret docker-registry.
- See the `imagePullSecrets` field within the container definitions of a Pod

Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup 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.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 `k8s.gcr.io/busybox` image. Here is the configuration file for the Pod:

pods/probe/exec-liveness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/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 seconds before performing the first probe. To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the target 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 apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

Within 30 seconds, view the Pod events:

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

```
Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  11s   default-scheduler  Successfully
assigned default/liveness-exec to node01
  Normal  Pulling    9s    kubelet, node01    Pulling image
"k8s.gcr.io/busybox"
  Normal  Pulled     7s    kubelet, node01    Successfully
pulled image "k8s.gcr.io/busybox"
  Normal  Created    7s    kubelet, node01    Created container
liveness
  Normal  Started    7s    kubelet, node01    Started container
liveness
```

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.

```
  Type     Reason     Age                From               
Message
  ----     ------     ----               ----               
-------
  Normal   Scheduled  57s                default-scheduler  
Successfully assigned default/liveness-exec to node01
  Normal   Pulling    55s                kubelet, node01    
Pulling image "k8s.gcr.io/busybox"
  Normal   Pulled     53s                kubelet, node01    
Successfully pulled image "k8s.gcr.io/busybox"
```

```
  Normal   Created    53s                   kubelet, node01
Created container liveness
  Normal   Started    53s                   kubelet, node01
Started container liveness
  Warning  Unhealthy  10s (x3 over 20s)  kubelet, node01
Liveness probe failed: cat: can't open '/tmp/healthy': No such
file or directory
  Normal   Killing    10s                   kubelet, node01
Container liveness failed liveness probe, will be restarted
```

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 `k8s.gcr.io/liveness` image.

pods/probe/http-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

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 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.R
equest) {
    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 apply -f https://k8s.io/examples/pods/probe/http-
liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the container has been restarted:

```
kubectl describe pod liveness-http
```

In releases prior to v1.13 (including v1.13), if the environment variable `http_proxy` (or `HTTP_PROXY`) is set on the node where a Pod is running, the HTTP liveness probe uses that proxy. In releases after v1.13, local HTTP proxy environment variable settings do not affect the HTTP liveness probe.

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.

pods/probe/tcp-liveness-readiness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/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. Similar to 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.

To try the TCP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/tcp-liveness-readiness.yaml
```

After 15 seconds, view Pod events to verify that liveness probes:

```
kubectl describe pod goproxy
```

Define a gRPC liveness probe

FEATURE STATE: `Kubernetes v1.23 [alpha]`

If your application implements gRPC Health Checking Protocol, kubelet can be configured to use it for application liveness checks. You must enable the `GRPCContainerProbe` feature gate in order to configure checks that rely on gRPC.

Here is an example manifest:

pods/probe/grpc-liveness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: etcd-with-grpc
spec:
  containers:
  - name: etcd
    image: k8s.gcr.io/etcd:3.5.1-0
    command: [ "/usr/local/bin/etcd", "--data-dir",  "/var/lib/etcd", "--listen-client-urls", "http://0.0.0.0:2379", "--advertise-client-urls", "http://127.0.0.1:2379", "--log-level", "debug"]
    ports:
    - containerPort: 2379
    livenessProbe:
      grpc:
        port: 2379
      initialDelaySeconds: 10
```

To use a gRPC probe, `port` must be configured. If the health endpoint is configured on a non-default service, you must also specify the `service`.

Note: Unlike HTTP and TCP probes, named ports cannot be used and custom host cannot be configured.

Configuration problems (for example: incorrect port and service, unimplemented health checking protocol) are considered a probe failure, similar to HTTP and TCP probes.

To try the gRPC liveness check, create a Pod using the command below. In the example below, the etcd pod is configured to use gRPC liveness probe.

```
kubectl apply -f https://k8s.io/examples/pods/probe/grpc-liveness.yaml
```

After 15 seconds, view Pod events to verify that the liveness check has not failed:

```
kubectl describe pod etcd-with-grpc
```

Before Kubernetes 1.23, gRPC health probes were often implemented using grpc-health-probe, as described in the blog post Health checking gRPC servers on Kubernetes. The built-in gRPC probes behavior is similar to one implemented by grpc-health-probe. When migrating from grpc-health-probe to built-in probes, remember the following differences:

- Built-in probes run against the pod IP address, unlike grpc-health-probe that often runs against `127.0.0.1`. Be sure to configure your gRPC endpoint to listen on the Pod's IP address.
- Built-in probes do not support any authentication parameters (like `-tls`).
- There are no error codes for built-in probes. All errors are considered as probe failures.
- If `ExecProbeTimeout` feature gate is set to `false`, grpc-health-probe does **not** respect the `timeoutSeconds` setting (which defaults to 1s), while built-in probe would fail on timeout.

Use a named port

You can use a named `port` for HTTP and TCP probes. (gRPC probes do not support named ports).

For example:

```yaml
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
```

Protect slow starting containers with startup probes

Sometimes, you have to deal with legacy applications that might require an additional startup time on their first initialization. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe. The trick is to set up a startup probe with the same command, HTTP or TCP check, with a `failureThreshold * periodSeconds` long enough to cover the worse case startup time.

So, the previous example would become:

```yaml
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

Thanks to the startup probe, the application will have a maximum of 5 minutes (30 * 10 = 300s) to finish its startup. Once the startup probe has succeeded once, the liveness probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's `restartPolicy`.

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, or depend on external services after 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.

Note: Readiness probes runs on the container during its whole lifecycle.
Caution: Liveness probes *do not* wait for readiness probes to succeed. If you want to wait before executing a liveness probe you should use initialDelaySeconds or a startupProbe.

Readiness probes are configured similarly to liveness probes. The only difference is that you use the `readinessProbe` field instead of the `livenessProbe` field.

```yaml
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 or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
- `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 and startup Probes. Minimum value is 1.
- `failureThreshold`: When a probe fails, Kubernetes will try `failureThreshold` times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

Note:

Before Kubernetes 1.20, the field `timeoutSeconds` was not respected for exec probes: probes continued running indefinitely, even past their configured deadline, until a result was returned.

This defect was corrected in Kubernetes v1.20. You may have been relying on the previous behavior, even without realizing it, as the default timeout is 1 second. As a cluster administrator, you can disable the feature gate `ExecProbeTimeout` (set it to `false`) on each kubelet to restore the behavior from older versions, then remove that override once all the exec probes in the cluster have a `timeoutSeconds` value set. If you have pods that are impacted from the default 1 second timeout, you should update their probe timeout so that you're ready for the eventual removal of that feature gate.

With the fix of the defect, for exec probes, on Kubernetes `1.20+` with the `dockershim` container runtime, the process inside the container may keep running even after probe returned failure because of the timeout.

Caution: Incorrect implementation of readiness probes may result in an ever growing number of processes in the container, and resource starvation if this is left unchecked.

HTTP probes

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. Defaults to /.
- `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 pod'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`.

For an HTTP probe, the kubelet sends two request headers in addition to the mandatory `Host` header: `User-Agent`, and `Accept`. The default values for these headers are `kube-probe/1.23` (where `1.23` is the version of the kubelet), and `*/*` respectively.

You can override the default headers by defining `.httpHeaders` for the probe; for example

```yaml
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: application/json

startupProbe:
  httpGet:
    httpHeaders:
      - name: User-Agent
        value: MyUserAgent
```

You can also remove these two headers by defining them with an empty value.

```yaml
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: ""
```

```yaml
startupProbe:
  httpGet:
    httpHeaders:
      - name: User-Agent
        value: ""
```

TCP probes

For a TCP probe, the kubelet makes the probe connection at the node, not in the pod, which means that you can not use a service name in the `host` parameter since the kubelet is unable to resolve it.

Probe-level `terminationGracePeriodSeconds`

FEATURE STATE: `Kubernetes v1.22 [beta]`

Prior to release 1.21, the pod-level `terminationGracePeriodSeconds` was used for terminating a container that failed its liveness or startup probe. This coupling was unintended and may have resulted in failed containers taking an unusually long time to restart when a pod-level `terminationGracePeriodSeconds` was set.

In 1.21 and beyond, when the feature gate `ProbeTerminationGracePeriod` is enabled, users can specify a probe-level `terminationGracePeriodSeconds` as part of the probe specification. When the feature gate is enabled, and both a pod- and probe-level `terminationGracePeriodSeconds` are set, the kubelet will use the probe-level value.

Note:

As of Kubernetes 1.22, the `ProbeTerminationGracePeriod` feature gate is only available on the API Server. The kubelet always honors the probe-level `terminationGracePeriodSeconds` field if it is present on a Pod.

If you have existing Pods where the `terminationGracePeriodSeconds` field is set and you no longer wish to use per-probe termination grace periods, you must delete those existing Pods.

When you (or the control plane, or some other component) create replacement Pods, and the feature gate `ProbeTerminationGracePeriod` is disabled, then the API server ignores the Pod-level `terminationGracePeriodSeconds` field, even if a Pod or pod template specifies it.

For example,

```yaml
spec:
  terminationGracePeriodSeconds: 3600  # pod-level
  containers:
  - name: test
    image: ...
    ports:
    - name: liveness-port
      containerPort: 8080
      hostPort: 8080

    livenessProbe:
      httpGet:
        path: /healthz
        port: liveness-port
      failureThreshold: 1
      periodSeconds: 60
      # Override pod-level terminationGracePeriodSeconds #
      terminationGracePeriodSeconds: 60
```

Probe-level `terminationGracePeriodSeconds` cannot be set for readiness probes. It will be rejected by the API server.

What's next

- Learn more about Container Probes.

You can also read the API references for:

- Pod, and specifically:
  - container(s)
  - probe(s)

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

```
kubectl get nodes --show-labels
```

The output is similar to this:

```
NAME      STATUS    ROLES    AGE     VERSION        LABELS
worker0   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker0
worker1   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker2
```

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    ROLES    AGE     VERSION        LABELS
worker0   Ready     <none>   1d
v1.13.0        ...,disktype=ssd,kubernetes.io/hostname=worker
0
worker1   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker2
```

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.

pods/pod-nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
```

```yaml
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 apply -f https://k8s.io/examples/pods/pod-nginx.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
   ```

Create a pod that gets scheduled to specific node

You can also schedule a pod to one specific node via setting `nodeName`.

pods/pod-nginx-specific-node.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: foo-node # schedule pod to specific node
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

Use the configuration file to create a pod that will get scheduled on `foo-node` only.

What's next

- Learn more about labels and selectors.
- Learn more about nodes.

Assign Pods to Nodes using Node Affinity

This page shows how to assign a Kubernetes Pod to a particular node using Node Affinity 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.

Add a label to a node

1. List the nodes in your cluster, along with their labels:

   ```
   kubectl get nodes --show-labels
   ```

   The output is similar to this:

   ```
   NAME      STATUS    ROLES    AGE     VERSION        LABELS
   worker0   Ready     <none>   1d
   v1.13.0        ...,kubernetes.io/hostname=worker0
   worker1   Ready     <none>   1d
   v1.13.0        ...,kubernetes.io/hostname=worker1
   worker2   Ready     <none>   1d
   v1.13.0        ...,kubernetes.io/hostname=worker2
   ```

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    ROLES    AGE     VERSION        LABELS
   worker0   Ready     <none>   1d
   ```

```
v1.13.0        ...,disktype=ssd,kubernetes.io/
hostname=worker0
worker1   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker1
worker2   Ready     <none>   1d
v1.13.0        ...,kubernetes.io/hostname=worker2
```

In the preceding output, you can see that the `worker0` node has a `disktype=ssd` label.

Schedule a Pod using required node affinity

This manifest describes a Pod that has a `requiredDuringSchedulingIgnoredDuringExecution` node affinity,`disktype: ssd`. This means that the pod will get scheduled only on a node that has a `disktype=ssd` label.

pods/pod-nginx-required-affinity.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

   ```
   kubectl apply -f https://k8s.io/examples/pods/pod-nginx-
   required-affinity.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
```

Schedule a Pod using preferred node affinity

This manifest describes a Pod that has a `preferredDuringSchedulingIgnoredDuringExecution` node affinity,`disktype: ssd`. This means that the pod will prefer a node that has a `disktype=ssd` label.

pods/pod-nginx-preferred-affinity.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

   ```
   kubectl apply -f https://k8s.io/examples/pods/pod-nginx-
   preferred-affinity.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 Node Affinity.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

pods/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:1.28
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://info.cern.ch
    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://info.cern.ch
```

Notice that the init container writes the `index.html` file in the root directory of the nginx server.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/init-
containers.yaml
```

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
init-demo   1/1       Running   0          1m
```

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:

```
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
  ...
  <li><a href="http://info.cern.ch/hypertext/WWW/
TheProject.html">Browse the first website</a></li>
  ...
```

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. A Container may specify one handler per event.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

pods/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/share/message"]
      preStop:
        exec:
          command: ["/bin/sh","-c","nginx -s quit; while killall
-0 nginx; do sleep 1; done"]
```

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 apply -f https://k8s.io/examples/pods/lifecycle-
events.yaml
```

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 Pod Lifecycle.

Note: Kubernetes only sends the preStop event when a Pod is *terminated*. This means that the preStop hook is not invoked when the Pod is *completed*. This limitation is tracked in issue #55087.

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 a Pod to Use a ConfigMap

Many applications rely on configuration which is used during either application initialization or runtime. Most of the times there is a requirement to adjust values assigned to configuration parameters. ConfigMaps is the kubernetes way to inject application pods with configuration data. ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. This page provides a series of usage examples demonstrating how to create ConfigMaps and 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Create a ConfigMap

You can use either `kubectl create configmap` or a ConfigMap generator in `kustomization.yaml` to create a ConfigMap. Note that `kubectl` starts to support `kustomization.yaml` since 1.14.

Create a ConfigMap Using kubectl create 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 name of a ConfigMap object must be a valid DNS subdomain name.

When you are creating a ConfigMap based on a file, the key in the <data-source> defaults to the basename of the file, and the value defaults to the file content.

You can use `kubectl describe` or `kubectl get` to retrieve information about a ConfigMap.

Create ConfigMaps from directories

You can use `kubectl create configmap` to create a ConfigMap from multiple files in the same directory. When you are creating a ConfigMap based on a directory, kubectl identifies files whose basename is a valid key in the directory and packages each of those files into the new ConfigMap. Any directory entries except regular files are ignored (e.g. subdirectories, symlinks, devices, pipes, etc).

For example:

```
# Create the local directory
mkdir -p configure-pod-container/configmap/

# Download the sample files into `configure-pod-container/
configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O
configure-pod-container/configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O
configure-pod-container/configmap/ui.properties

# Create the configmap
kubectl create configmap game-config --from-file=configure-pod-
container/configmap/
```

The above command packages each file, in this case, `game.properties` and `ui.properties` in the `configure-pod-container/configmap/` directory into the game-config ConfigMap. You can display details of the ConfigMap using the following command:

```
kubectl describe configmaps game-config
```

The output is similar to this:

```
Name:          game-config
Namespace:     default
Labels:        <none>
Annotations:   <none>

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
```

The `game.properties` and `ui.properties` files in the `configure-pod-container/configmap/` directory are represented in the `data` section of the ConfigMap.

```
kubectl get configmaps game-config -o yaml
```

The output is similar to this:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
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
```

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=configure-pod-
container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
```

where the output is similar to this:

```
Name:         game-config-2
Namespace:    default
Labels:       <none>
Annotations:  <none>

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
```

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=configure-pod-
container/configmap/game.properties --from-file=configure-pod-
container/configmap/ui.properties
```

You can display details of the `game-config-2` ConfigMap using the following command:

```
kubectl describe configmaps game-config-2
```

The output is similar to this:

```
Name:          game-config-2
Namespace:     default
Labels:        <none>
Annotations:   <none>

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
```

When `kubectl` creates a ConfigMap from inputs that are not ASCII or UTF-8, the tool puts these into the `binaryData` field of the ConfigMap, and not in `data`. Both text and binary data sources can be combined in one ConfigMap. If you want to view the `binaryData` keys (and their values) in a ConfigMap, you can run `kubectl get configmap -o jsonpath='{.binaryData}' <name>`.

Use the option `--from-env-file` to create a ConfigMap from an env-file, for example:

```
# Env-files contain a list of environment variables.
# These syntax rules apply:
#   Each line in an env file has to be in VAR=VAL format.
#   Lines beginning with # (i.e. comments) are ignored.
#   Blank lines are ignored.
#   There is no special handling of quotation marks (i.e. they
will be part of the ConfigMap value)).

# Download the sample files into `configure-pod-container/
configmap/` directory
wget https://kubernetes.io/examples/configmap/game-env-
file.properties -O configure-pod-container/configmap/game-env-
file.properties
wget https://kubernetes.io/examples/configmap/ui-env-
file.properties -O configure-pod-container/configmap/ui-env-
file.properties

# The env-file `game-env-file.properties` looks like below
cat configure-pod-container/configmap/game-env-file.properties
enemies=aliens
lives=3
allowed="true"

# This comment and the empty line above it are ignored
```

```
kubectl create configmap game-config-env-file \
       --from-env-file=configure-pod-container/configmap/game-
env-file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap game-config-env-file -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:36:28Z
  name: game-config-env-file
  namespace: default
  resourceVersion: "809965"
  uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
```

Starting with Kubernetes v1.23, `kubectl` supports the `--from-env-file` argument to be specified multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap config-multi-env-files \
        --from-env-file=configure-pod-container/configmap/game-
env-file.properties \
        --from-env-file=configure-pod-container/configmap/ui-env-
file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap config-multi-env-files -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
data:
  allowed: '"true"'
  color: purple
  enemies: aliens
  how: fairlyNice
  lives: "3"
  textmode: "true"
```

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=configure-pod-container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl get configmaps game-config-3 -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
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
```

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.ho
w=very --from-literal=special.type=charm
```

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
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  uid: dadce046-d673-11e5-8cd0-68f728db1985
data:
  special.how: very
  special.type: charm
```

Create a ConfigMap from generator

`kubectl` supports `kustomization.yaml` since 1.14. You can also create a ConfigMap from generators and then apply it to create the object on the Apiserver. The generators should be specified in a `kustomization.yaml` inside a directory.

Generate ConfigMaps from files

For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-4
  files:
  - configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/game-config-4-m9dm2f92bt created
```

You can check that the ConfigMap was created like this:

```
kubectl get configmap
NAME                       DATA   AGE
game-config-4-m9dm2f92bt   1      37s


kubectl describe configmaps/game-config-4-m9dm2f92bt
Name:         game-config-4-m9dm2f92bt
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","data":{"game.properties":"ene
mies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noG
oodRotten\nsecret.code.p...

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
Events:  <none>
```

Note that the generated ConfigMap name has a suffix appended by hashing the contents. This ensures that a new ConfigMap is generated each time the content is modified.

Define the key to use when generating a ConfigMap from a file

You can define a key other than the file name to use in the ConfigMap generator. For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties` with the key `game-special-key`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-5
  files:
  - game-special-key=configure-pod-container/configmap/
game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/game-config-5-m67dt67794 created
```

Generate ConfigMaps from Literals

To generate a ConfigMap from literals `special.type=charm` and `special.how=very`, you can specify the ConfigMap generator in `kustomization.yaml` as

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: special-config-2
  literals:
  - special.how=very
  - special.type=charm
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/special-config-2-c92b5mmcf2 created
```

Define container environment variables using ConfigMap data

Define a container 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.

   pods/pod-single-configmap-env-variable.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/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 SPECIAL_LEVEL_KEY
              name: special-config
              # Specify the key associated with the value
              key: special.how
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-single-configmap-env-variable.yaml
```

Now, the Pod's output includes environment variable `SPECIAL_LEVEL_KEY=very`.

Define container environment variables with data from multiple ConfigMaps

- As with the previous example, create the ConfigMaps first.

[configmap/configmaps.yaml]

```
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
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmaps.yaml
```

- Define the environment variables in the Pod specification.

[pods/pod-multiple-configmap-env-variable.yaml]

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
```

```
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/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
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-multiple-configmap-env-variable.yaml
```

Now, the Pod's output includes environment variables `SPECIAL_LEVEL_KEY=very` and `LOG_LEVEL=INFO`.

Configure all key-value pairs in a ConfigMap as container environment variables

Note: This functionality is available in Kubernetes v1.6 and later.

- Create a ConfigMap containing multiple key-value pairs.

configmap/configmap-multikeys.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/
configmap-multikeys.yaml
```

- Use `envFrom` to define all of the ConfigMap's data as container environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.

pods/pod-configmap-envFrom.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - configMapRef:
          name: special-config
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-
configmap-envFrom.yaml
```

Now, the Pod's output includes environment variables `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` and `args` of a container using the `$(VAR_NAME)` Kubernetes substitution syntax.

For example, the following Pod specification

pods/pod-configmap-env-var-valueFrom.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/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
```

created by running

```
kubectl create -f https://kubernetes.io/examples/pods/pod-
configmap-env-var-valueFrom.yaml
```

produces the following output in the `test-container` container:

```
very charm
```

Add ConfigMap data to a Volume

As explained in [Create ConfigMaps from files](), 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.

[`configmap/configmap-multikeys.yaml`]()

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/
configmap-multikeys.yaml
```

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 lists directory files with names that match the keys in ConfigMap.

[pods/pod-configmap-volume.yaml]

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

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-
configmap-volume.yaml
```

When the pod runs, the command `ls /etc/config/` produces the output below:

```
SPECIAL_LEVEL
SPECIAL_TYPE
```

Caution: If there are some files in the `/etc/config/` directory, they will be deleted.
Note: Text data is exposed as files using the UTF-8 character encoding. To use some other character encoding, use binaryData.

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`.

[pods/pod-configmap-volume-specific-key.yaml]()

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

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume-specific-key.yaml
```

When the pod runs, the command `cat /etc/config/keys` produces the output below:

```
very
```

Caution: Like before, all previous files in the `/etc/config/` directory will be deleted.

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.

Optional References

A ConfigMap reference may be marked "optional". If the ConfigMap is non-existent, the mounted volume will be empty. If the ConfigMap exists, but the referenced key is non-existent the path will be absent beneath the mount point.

Mounted ConfigMaps are updated automatically

When a mounted ConfigMap is updated, the projected content is eventually updated too. This applies in the case where an optionally referenced ConfigMap comes into existence after a pod has started.

Kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, it uses 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 (1 minute by default) + TTL of ConfigMaps cache (1 minute by default) in kubelet. You can trigger an immediate refresh by updating one of the pod's annotations.

Note: A container using a ConfigMap as a subPath volume will not receive ConfigMap updates.

Understanding ConfigMaps and Pods

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`.

```yaml
apiVersion: v1
kind: ConfigMap
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
```

Restrictions

- 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.

- 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
  ```

  The output is similar to this:

  ```
  LASTSEEN FIRSTSEEN COUNT NAME          KIND  SUBOBJECT
  TYPE      REASON
  SOURCE                MESSAGE
  0s        0s        1     dapi-test-pod Pod
  Warning   InvalidEnvironmentVariableNames   {kubelet,
  127.0.0.1}  Keys [1badkey, 2alsobad] from the EnvFrom
  configMap default/myconfig were skipped since they are
  considered invalid environment variable names.
  ```

- ConfigMaps reside in a specific Namespace. A ConfigMap can only be referenced by pods residing in the same namespace.

- You can't use ConfigMaps for static pods, because the Kubelet does not support this.

What's next

- Follow a real world example of Configuring Redis using a ConfigMap.

Share Process Namespace between Containers in a Pod

FEATURE STATE: `Kubernetes v1.17 [stable]`

This page shows how to configure process namespace sharing for a pod. When process namespace sharing is enabled, processes in a container are visible to all other containers in that pod.

You can use this feature to configure cooperating containers, such as a log handler sidecar container, or to troubleshoot container images that don't include debugging utilities like a shell.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.

Configure a Pod

Process Namespace Sharing is enabled using the `shareProcessNamespace` field of `v1.PodSpec`. For example:

pods/share-process-namespace.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox:1.28
    securityContext:
      capabilities:
        add:
        - SYS_PTRACE
    stdin: true
    tty: true
```

1. Create the pod `nginx` on your cluster:

   ```
   kubectl apply -f https://k8s.io/examples/pods/share-process-namespace.yaml
   ```

2. Attach to the `shell` container and run `ps`:

   ```
   kubectl attach -it nginx -c shell
   ```

   If you don't see a command prompt, try pressing enter.

```
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    8 root      0:00 nginx: master process nginx -g daemon
off;
   14 101       0:00 nginx: worker process
   15 root      0:00 sh
   21 root      0:00 ps ax
```

You can signal processes in other containers. For example, send `SIGHUP` to nginx to restart the worker process. This requires the `SYS_PTRACE` capability.

```
/ # kill -HUP 8
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    8 root      0:00 nginx: master process nginx -g daemon off;
   15 root      0:00 sh
   22 101       0:00 nginx: worker process
   23 root      0:00 ps ax
```

It's even possible to access another container image using the `/proc/$pid/root` link.

```
/ # head /proc/8/root/etc/nginx/nginx.conf

user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
```

Understanding Process Namespace Sharing

Pods share many resources so it makes sense they would also share a process namespace. Some container images may expect to be isolated from other containers, though, so it's important to understand these differences:

1. **The container process no longer has PID 1.** Some container images refuse to start without PID 1 (for example, containers using `systemd`) or run commands like `kill -HUP 1` to signal the container process. In pods with a shared process namespace, `kill -HUP 1` will signal the pod sandbox. (`/pause` in the above example.)

2. **Processes are visible to other containers in the pod.** This includes all information visible in `/proc`, such as passwords that were passed as arguments or environment variables. These are protected only by regular Unix permissions.

3. **Container filesystems are visible to other containers in the pod through the `/proc/$pid/root` link.** This makes debugging easier, but it also means that filesystem secrets are protected only by filesystem permissions.

Create static Pods

Static Pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. Unlike Pods that are managed by the control plane (for example, a Deployment); instead, the kubelet watches each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there. The Pod names will be suffixed with the node hostname with a leading hyphen.

Note: 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 instead.
Note: The `spec` of a static Pod cannot refer to other API objects (e.g., ServiceAccount, ConfigMap, Secret, etc).

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

This page assumes you're using CRI-O to run Pods, and that your nodes are running the Fedora operating system. Instructions for other distributions or Kubernetes installations may vary.

Create a static pod

You can configure a static Pod with either a file system hosted configuration file or a web hosted configuration file.

Filesystem-hosted static Pod manifest

Manifests are standard Pod definitions in JSON or YAML format in a specific directory. Use the `staticPodPath: <the directory>` field in the kubelet configuration file, which periodically scans the directory and creates/deletes static Pods as YAML/JSON files appear/disappear there. Note that the 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 you want to run the static Pod. In this example, it's `my-node1`.

   ```
   ssh my-node1
   ```

2. Choose a directory, say `/etc/kubelet.d` and place a web server Pod definition there, for example `/etc/kubelet.d/static-web.yaml`:

   ```
   # Run this command on the node where kubelet is running
   mkdir /etc/kubelet.d/
   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 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/"
   ```

   or add the `staticPodPath: <the directory>` field in the kubelet configuration file.

4. Restart the kubelet. On Fedora, you would run:

   ```
   # Run this command on the node where the kubelet is running
   systemctl restart kubelet
   ```

Web-hosted static pod manifest

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a JSON/YAML file that contains Pod definitions. Similar to how filesystem-hosted manifests work, the kubelet refetches the manifest on a schedule. If there are changes to the list of static Pods, the kubelet applies them.

To use this approach:

1. Create a YAML file and store it on a web server so that you can pass the URL of that file to the kubelet.

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: static-web
     labels:
       role: myrole
   spec:
     containers:
       - name: web
         image: nginx
         ports:
           - name: web
             containerPort: 80
             protocol: TCP
   ```

2. Configure the kubelet on your selected node to use this web manifest by running it with `--manifest-url=<manifest-url>`. On Fedora, edit `/etc/kubernetes/kubelet` to include this line:

   ```
   KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-
   domain=kube.local --manifest-url=<manifest-url>"
   ```

3. Restart the kubelet. On Fedora, you would run:

   ```
   # Run this command on the node where the kubelet is running
   systemctl restart kubelet
   ```

Observe static pod behavior

When the kubelet starts, it automatically starts all defined static Pods. As you have defined a static Pod and restarted the kubelet, the new static Pod should already be running.

You can view running containers (including static Pods) by running (on the node):

```
# Run this command on the node where the kubelet is running
crictl ps
```

The output might be something like:

```
CONTAINER       IMAGE
CREATED           STATE      NAME     ATTEMPT    POD ID
129fd7d382018   docker.io/library/nginx@sha256:...    11 minutes
ago    Running    web     0          34533c6729106
```

Note: `crictl` outputs the image URI and SHA-256 checksum. `NAME` will look more like: `docker.io/library/nginx@sha256:0d17b565c37bcbd895e9d92315a05c1c3c9a29f762b011a10c54a66cd53c9b31`.

You can see the mirror Pod on the API server:

```
kubectl get pods
```

```
NAME          READY   STATUS    RESTARTS        AGE
static-web    1/1     Running   0               2m
```

Note: Make sure the kubelet has permission to create the mirror Pod in the API server. If not, the creation request is rejected by the API server. See Pod Security admission and PodSecurityPolicy.

Labels from the static Pod are propagated into the mirror Pod. You can use those labels as normal via selectors, etc.

If you try to use `kubectl` to delete the mirror Pod from the API server, the kubelet *doesn't* remove the static Pod:

```
kubectl delete pod static-web
```

```
pod "static-web" deleted
```

You can see that the Pod is still running:

```
kubectl get pods
```

```
NAME          READY   STATUS    RESTARTS   AGE
static-web    1/1     Running   0          4s
```

Back on your node where the kubelet is running, you can try to stop the container manually. You'll see that, after a time, the kubelet will notice and will restart the Pod automatically:

```
# Run these commands on the node where the kubelet is running
crictl stop 129fd7d382018 # replace with the ID of your container
sleep 20
crictl ps
```

```
CONTAINER       IMAGE
CREATED           STATE      NAME     ATTEMPT    POD ID
89db4553e1eeb   docker.io/library/nginx@sha256:...    19 seconds
ago    Running    web     1          34533c6729106
```

Dynamic addition and removal of static pods

The 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.

```
# This assumes you are using filesystem-hosted static Pod
configuration
# Run these commands on the node where the kubelet is running
#
mv /etc/kubelet.d/static-web.yaml /tmp
sleep 20
crictl ps
# You see that no nginx container is running
mv /tmp/static-web.yaml  /etc/kubelet.d/
sleep 20
crictl ps
```

```
CONTAINER       IMAGE
CREATED           STATE      NAME    ATTEMPT    POD ID
f427638871c35   docker.io/library/nginx@sha256:...    19 seconds
ago    Running    web     1          34533c6729106
```

Translate a Docker Compose File to Kubernetes Resources

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 on the Kompose website at http://kompose.io.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Install Kompose

We have multiple ways to install Kompose. Our preferred method is downloading the binary from the latest GitHub release.

- GitHub download
- Build from source
- CentOS package
- Fedora package
- Homebrew (macOS)

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/
v1.26.0/kompose-linux-amd64 -o kompose

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/
v1.26.0/kompose-darwin-amd64 -o kompose

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/
v1.26.0/kompose-windows-amd64.exe -o kompose.exe

chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Alternatively, you can download the tarball.

Installing using `go get` pulls from the master branch with the latest development changes.

```
go get -u github.com/kubernetes/kompose
```

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
```

Kompose is in Fedora 24, 25 and 26 repositories. You can install it like any other package.

```
sudo dnf -y install kompose
```

On macOS you can install latest release via Homebrew:

```
brew install kompose
```

Use Kompose

In a few steps, we'll take you from Docker Compose to Kubernetes. All you need is an existing `docker-compose.yml` file.

1. Go to the directory containing your `docker-compose.yml` file. If you don't have one, test using this one.

   ```
   version: "2"

   services:

     redis-master:
       image: k8s.gcr.io/redis:e2e
       ports:
         - "6379"

     redis-slave:
       image: gcr.io/google_samples/gb-redisslave:v3
       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. To convert the `docker-compose.yml` file to files that you can use with `kubectl`, run `kompose convert` and then `kubectl apply -f <output file>`.

   ```
   kompose convert
   ```

   The output is similar to:

   ```
   INFO Kubernetes file "frontend-service.yaml" created
      INFO Kubernetes file "frontend-service.yaml" created
   INFO Kubernetes file "frontend-service.yaml" created
   INFO Kubernetes file "redis-master-service.yaml" created
      INFO Kubernetes file "redis-master-service.yaml" created
   INFO Kubernetes file "redis-master-service.yaml" created
   INFO Kubernetes file "redis-slave-service.yaml" created
      INFO Kubernetes file "redis-slave-service.yaml" created
   INFO Kubernetes file "redis-slave-service.yaml" created
   INFO Kubernetes file "frontend-deployment.yaml" created
      INFO Kubernetes file "frontend-deployment.yaml" created
   ```

```
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
   INFO Kubernetes file "redis-master-deployment.yaml"
created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
   INFO Kubernetes file "redis-slave-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

```
 kubectl apply -f frontend-service.yaml,redis-master-
service.yaml,redis-slave-service.yaml,frontend-
deployment.yaml,redis-master-deployment.yaml,redis-slave-
deployment.yaml
```

The output is similar to:

```
service/frontend created
service/redis-master created
service/redis-slave created
deployment.apps/frontend created
deployment.apps/redis-master created
deployment.apps/redis-slave created
```

Your deployments are running in Kubernetes.

3. Access your application.

   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:   192.0.2.89
   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://192.0.2.89
   ```

User Guide

- CLI
  - `kompose convert`
- Documentation
  - 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 convert` example

```
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.yaml  result-deployment.yaml  vote-
deployment.yaml  worker-deployment.yaml
db-svc.yaml         docker-voting.yml          redis-
svc.yaml     result-svc.yaml        vote-svc.yaml
worker-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-
service.jsonmlbparks-service.yaml
frontend-deployment.yaml  mongodb-claim0-
persistentvolumeclaim.yaml  redis-master-service.yaml
frontend-service.yaml     mongodb-
deployment.yaml                     redis-slave-deployment.yaml
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 convert` example

```
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 source.
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 .

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, Daemon 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 DaemonSet objects

If you want to generate a Chart to be used with Helm run:

```
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
â”œâ”€â”€ Chart.yaml
â”œâ”€â”€ README.md
â””â”€â”€ templates
    â”œâ”€â”€ redis-deployment.yaml
    â”œâ”€â”€ redis-svc.yaml
    â”œâ”€â”€ web-deployment.yaml
    â””â”€â”€ 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:

```yaml
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: The `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 happens 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: The controller object could be `deployment` or `replicationcontroller`.

For example, the `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 Configurations

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.

Enforce Pod Security Standards by Configuring the Built-in Admission Controller

As of v1.22, Kubernetes provides a built-in admission controller to enforce the Pod Security Standards. You can configure this admission controller to set cluster-wide defaults and exemptions.

Before you begin

Your Kubernetes server must be at or later than version v1.22. To check the version, enter `kubectl version`.

- Ensure the `PodSecurity` feature gate is enabled.

Configure the Admission Controller

- pod-security.admission.config.k8s.io/v1beta1
- pod-security.admission.config.k8s.io/v1alpha1

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
```

```yaml
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1beta1
    kind: PodSecurityConfiguration
    # Defaults applied when a mode label is not set.
    #
    # Level label values must be one of:
    # - "privileged" (default)
    # - "baseline"
    # - "restricted"
    #
    # Version label values must be one of:
    # - "latest" (default)
    # - specific version like "v1.23"
    defaults:
      enforce: "privileged"
      enforce-version: "latest"
      audit: "privileged"
      audit-version: "latest"
      warn: "privileged"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: []
```

Note: v1beta1 configuration requires v1.23+. For v1.22, use v1alpha1.

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1alpha1
    kind: PodSecurityConfiguration
    # Defaults applied when a mode label is not set.
    #
    # Level label values must be one of:
    # - "privileged" (default)
    # - "baseline"
    # - "restricted"
    #
    # Version label values must be one of:
    # - "latest" (default)
    # - specific version like "v1.23"
    defaults:
      enforce: "privileged"
      enforce-version: "latest"
      audit: "privileged"
      audit-version: "latest"
      warn: "privileged"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: []
```

Enforce Pod Security Standards with Namespace Labels

Namespaces can be labeled to enforce the Pod Security Standards.

Before you begin

Your Kubernetes server must be at or later than version v1.22. To check the version, enter `kubectl version`.

- Ensure the `PodSecurity` feature gate is enabled.

Requiring the `baseline` Pod Security Standard with namespace labels

This manifest defines a Namespace `my-baseline-namespace` that:

- *Blocks* any pods that don't satisfy the `baseline` policy requirements.
- Generates a user-facing warning and adds an audit annotation to any created pod that does not meet the `restricted` policy requirements.
- Pins the versions of the `baseline` and `restricted` policies to v1.23.

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-baseline-namespace
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.23

    # We are setting these to our _desired_ `enforce` level.
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: v1.23
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: v1.23
```

Add labels to existing namespaces with `kubectl label`

Note: When an `enforce` policy (or version) label is added or changed, the admission plugin will test each pod in the namespace against the new policy. Violations are returned to the user as warnings.

It is helpful to apply the `--dry-run` flag when initially evaluating security profile changes for namespaces. The Pod Security Standard checks will still be run in *dry run* mode, giving you information about how the new policy would treat existing pods, without actually updating a policy.

```
kubectl label --dry-run=server --overwrite ns --all \
    pod-security.kubernetes.io/enforce=baseline
```

Applying to all namespaces

If you're just getting started with the Pod Security Standards, a suitable first step would be to configure all namespaces with audit annotations for a stricter level such as `baseline`:

```
kubectl label --overwrite ns --all \
  pod-security.kubernetes.io/audit=baseline \
  pod-security.kubernetes.io/warn=baseline
```

Note that this is not setting an enforce level, so that namespaces that haven't been explicitly evaluated can be distinguished. You can list namespaces without an explicitly set enforce level using this command:

```
kubectl get namespaces --selector='!pod-security.kubernetes.io/enforce'
```

Applying to a single namespace

You can update a specific namespace as well. This command adds the `enforce=restricted` policy to `my-existing-namespace`, pinning the restricted policy version to v1.23.

```
kubectl label --overwrite ns my-existing-namespace \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/enforce-version=v1.23
```

Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller

This page describes the process of migrating from PodSecurityPolicies to the built-in PodSecurity admission controller. This can be done effectively using a combination of dry-run and `audit` and `warn` modes, although this becomes harder if mutating PSPs are used.

Before you begin

Your Kubernetes server must be at or later than version v1.22. To check the version, enter `kubectl version`.

- Ensure the `PodSecurity` feature gate is enabled.

This page assumes you are already familiar with the basic Pod Security Admission concepts.

Overall approach

There are multiple strategies you can take for migrating from PodSecurityPolicy to Pod Security Admission. The following steps are one possible migration path, with a goal of minimizing both the risks of a production outage and of a security gap.

1. Decide whether Pod Security Admission is the right fit for your use case.
2. Review namespace permissions
3. Simplify & standardize PodSecurityPolicies
4. Update namespaces
   1. Identify an appropriate Pod Security level
   2. Verify the Pod Security level
   3. Enforce the Pod Security level
   4. Bypass PodSecurityPolicy
5. Review namespace creation processes
6. Disable PodSecurityPolicy

0. Decide whether Pod Security Admission is right for you

Pod Security Admission was designed to meet the most common security needs out of the box, and to provide a standard set of security levels across clusters. However, it is less flexible than PodSecurityPolicy. Notably, the following features are supported by PodSecurityPolicy but not Pod Security Admission:

- **Setting default security constraints** - Pod Security Admission is a non-mutating admission controller, meaning it won't modify pods before validating them. If you were relying on this aspect of PSP, you will need to either modify your workloads to meet the Pod Security constraints, or use a Mutating Admission Webhook to make those changes. See Simplify & Standardize PodSecurityPolicies below for more detail.
- **Fine-grained control over policy definition** - Pod Security Admission only supports 3 standard levels. If you require more control over

specific constraints, then you will need to use a Validating Admission Webhook to enforce those policies.
- **Sub-namespace policy granularity** - PodSecurityPolicy lets you bind different policies to different Service Accounts or users, even within a single namespace. This approach has many pitfalls and is not recommended, but if you require this feature anyway you will need to use a 3rd party webhook instead. The exception to this is if you only need to completely exempt specific users or RuntimeClasses, in which case Pod Security Admission does expose some static configuration for exemptions.

Even if Pod Security Admission does not meet all of your needs it was designed to be *complementary* to other policy enforcement mechanisms, and can provide a useful fallback running alongside other admission webhooks.

1. Review namespace permissions

Pod Security Admission is controlled by labels on namespaces. This means that anyone who can update (or patch or create) a namespace can also modify the Pod Security level for that namespace, which could be used to bypass a more restrictive policy. Before proceeding, ensure that only trusted, privileged users have these namespace permissions. It is not recommended to grant these powerful permissions to users that shouldn't have elevated permissions, but if you must you will need to use an admission webhook to place additional restrictions on setting Pod Security labels on Namespace objects.

2. Simplify & standardize PodSecurityPolicies

In this section, you will reduce mutating PodSecurityPolicies and remove options that are outside the scope of the Pod Security Standards. You should make the changes recommended here to an offline copy of the original PodSecurityPolicy being modified. The cloned PSP should have a different name that is alphabetically before the original (for example, prepend a `0` to it). Do not create the new policies in Kubernetes yet - that will be covered in the Rollout the updated policies section below.

2.a. Eliminate purely mutating fields

If a PodSecurityPolicy is mutating pods, then you could end up with pods that don't meet the Pod Security level requirements when you finally turn PodSecurityPolicy off. In order to avoid this, you should eliminate all PSP mutation prior to switching over. Unfortunately PSP does not cleanly separate mutating & validating fields, so this is not a straightforward migration.

You can start by eliminating the fields that are purely mutating, and don't have any bearing on the validating policy. These fields (also listed in the Mapping PodSecurityPolicies to Pod Security Standards reference) are:

- `.spec.defaultAllowPrivilegeEscalation`

- `.spec.runtimeClass.defaultRuntimeClassName`
- `.metadata.annotations['seccomp.security.alpha.kubernetes.io/defaultProfileName']`
- `.metadata.annotations['apparmor.security.beta.kubernetes.io/defaultProfileName']`
- `.spec.defaultAddCapabilities` - Although technically a mutating & validating field, these should be merged into `.spec.allowedCapabilities` which performs the same validation without mutation.

Caution: Removing these could result in workloads missing required configuration, and cause problems. See Rollout the updated policies below for advice on how to roll these changes out safely.

2.b. Eliminate options not covered by the Pod Security Standards

There are several fields in PodSecurityPolicy that are not covered by the Pod Security Standards. If you must enforce these options, you will need to supplement Pod Security Admission with an admission webhook, which is outside the scope of this guide.

First, you can remove the purely validating fields that the Pod Security Standards do not cover. These fields (also listed in the Mapping PodSecurityPolicies to Pod Security Standards reference with "no opinion") are:

- `.spec.allowedHostPaths`
- `.spec.allowedFlexVolumes`
- `.spec.allowedCSIDrivers`
- `.spec.forbiddenSysctls`
- `.spec.runtimeClass`

You can also remove the following fields, that are related to POSIX / UNIX group controls.

Caution: If any of these use the `MustRunAs` strategy they may be mutating! Removing these could result in workloads not setting the required groups, and cause problems. See Rollout the updated policies below for advice on how to roll these changes out safely.

- `.spec.runAsGroup`
- `.spec.supplementalGroups`
- `.spec.fsGroup`

The remaining mutating fields are required to properly support the Pod Security Standards, and will need to be handled on a case-by-case basis later:

- `.spec.requiredDropCapabilities` - Required to drop `ALL` for the Restricted profile.
- `.spec.seLinux` - (Only mutating with the `MustRunAs` rule) required to enforce the SELinux requirements of the Baseline & Restricted profiles.

- `.spec.runAsUser` - (Non-mutating with the `RunAsAny` rule) required to enforce `RunAsNonRoot` for the Restricted profile.
- `.spec.allowPrivilegeEscalation` - (Only mutating if set to `false`) required for the Restricted profile.

2.c. Rollout the updated PSPs

Next, you can rollout the updated policies to your cluster. You should proceed with caution, as removing the mutating options may result in workloads missing required configuration.

For each updated PodSecurityPolicy:

1. Identify pods running under the original PSP. This can be done using the `kubernetes.io/psp` annotation. For example, using kubectl:

   ```
   PSP_NAME="original" # Set the name of the PSP you're checking for
   kubectl get pods --all-namespaces -o jsonpath="{range .items[?(@.metadata.annotations.kubernetes\.io\/psp=='$PSP_NAME')]}{.metadata.namespace} {.metadata.name}{'\n'}{end}"
   ```

2. Compare these running pods against the original pod spec to determine whether PodSecurityPolicy has modified the pod. For pods created by a workload resource you can compare the pod with the PodTemplate in the controller resource. If any changes are identified, the original Pod or PodTemplate should be updated with the desired configuration. The fields to review are:
   - `.metadata.annotations['container.apparmor.security.beta.kubernetes.io/*']` (replace * with each container name)
   - `.spec.runtimeClassName`
   - `.spec.securityContext.fsGroup`
   - `.spec.securityContext.seccompProfile`
   - `.spec.securityContext.seLinuxOptions`
   - `.spec.securityContext.supplementalGroups`
   - On containers, under `.spec.containers[*]` and `.spec.initContainers[*]`:
     - `.securityContext.allowPrivilegeEscalation`
     - `.securityContext.capabilities.add`
     - `.securityContext.capabilities.drop`
     - `.securityContext.readOnlyRootFilesystem`
     - `.securityContext.runAsGroup`
     - `.securityContext.runAsNonRoot`
     - `.securityContext.runAsUser`
     - `.securityContext.seccompProfile`
     - `.securityContext.seLinuxOptions`
3. Create the new PodSecurityPolicies. If any Roles or ClusterRoles are granting `use` on all PSPs this could cause the new PSPs to be used instead of their mutating counter-parts.
4. Update your authorization to grant access to the new PSPs. In RBAC this means updating any Roles or ClusterRoles that grant the `use` permision on the original PSP to also grant it to the updated PSP.

5. Verify: after some soak time, rerun the command from step 1 to see if any pods are still using the original PSPs. Note that pods need to be recreated after the new policies have been rolled out before they can be fully verified.
6. (optional) Once you have verified that the original PSPs are no longer in use, you can delete them.

3. Update Namespaces

The following steps will need to be performed on every namespace in the cluster. Commands referenced in these steps use the `$NAMESPACE` variable to refer to the namespace being updated.

3.a. Identify an appropriate Pod Security level

Start reviewing the Pod Security Standards and familiarizing yourself with the 3 different levels.

There are several ways to choose a Pod Security level for your namespace:

1. **By security requirements for the namespace** - If you are familiar with the expected access level for the namespace, you can choose an appropriate level based on those requirements, similar to how one might approach this on a new cluster.
2. **By existing PodSecurityPolicies** - Using the Mapping PodSecurityPolicies to Pod Security Standards reference you can map each PSP to a Pod Security Standard level. If your PSPs aren't based on the Pod Security Standards, you may need to decide between choosing a level that is at least as permissive as the PSP, and a level that is at least as restrictive. You can see which PSPs are in use for pods in a given namespace with this command:

   ```
   kubectl get pods -n $NAMESPACE -o jsonpath="{.items[*].metada
   ta.annotations.kubernetes\.io\/psp}" | tr " " "\n" | sort -u
   ```

3. **By existing pods** - Using the strategies under Verify the Pod Security level, you can test out both the Baseline and Restricted levels to see whether they are sufficiently permissive for existing workloads, and chose the least-privileged valid level.

Caution: Options 2 & 3 above are based on *existing* pods, and may miss workloads that aren't currently running, such as CronJobs, scale-to-zero workloads, or other workloads that haven't rolled out.

3.b. Verify the Pod Security level

Once you have selected a Pod Security level for the namespace (or if you're trying several), it's a good idea to test it out first (you can skip this step if using the Privileged level). Pod Security includes several tools to help test and safely roll out profiles.

First, you can dry-run the policy, which will evaluate pods currently running in the namespace against the applied policy, without making the new policy take effect:

```
# $LEVEL is the level to dry-run, either "baseline" or
"restricted".
kubectl label --dry-run=server --overwrite ns $NAMESPACE pod-
security.kubernetes.io/enforce=$LEVEL
```

This command will return a warning for any *existing* pods that are not valid under the proposed level.

The second option is better for catching workloads that are not currently running: audit mode. When running under audit-mode (as opposed to enforcing), pods that violate the policy level are recorded in the audit logs, which can be reviewed later after some soak time, but are not forbidden. Warning mode works similarly, but returns the warning to the user immediately. You can set the audit level on a namespace with this command:

```
kubectl label --overwrite ns $NAMESPACE pod-
security.kubernetes.io/audit=$LEVEL
```

If either of these approaches yield unexpected violations, you will need to either update the violating workloads to meet the policy requirements, or relax the namespace Pod Security level.

3.c. Enforce the Pod Security level

When you are satisfied that the chosen level can safely be enforced on the namespace, you can update the namespace to enforce the desired level:

```
kubectl label --overwrite ns $NAMESPACE pod-
security.kubernetes.io/enforce=$LEVEL
```

3.d. Bypass PodSecurityPolicy

Finally, you can effectively bypass PodSecurityPolicy at the namespace level by binding the fully privileged PSP to all service accounts in the namespace.

```
# The following cluster-scoped commands are only needed once.
kubectl apply -f privileged-psp.yaml
kubectl create clusterrole privileged-psp --verb use --resource
podsecuritypolicies.policy --resource-name privileged

# Per-namespace disable
kubectl create -n $NAMESPACE rolebinding disable-psp --
clusterrole privileged-psp --group system:serviceaccounts:$NAMESP
ACE
```

Since the privileged PSP is non-mutating, and the PSP admission controller always prefers non-mutating PSPs, this will ensure that pods in this namespace are no longer being modified or restricted by PodSecurityPolicy.

The advantage to disabling PodSecurityPolicy on a per-namespace basis like this is if a problem arises you can easily roll the change back by deleting the RoleBinding. Just make sure the pre-existing PodSecurityPolicies are still in place!

```
# Undo PodSecurityPolicy disablement.
kubectl delete -n $NAMESPACE rolebinding disable-psp
```

4. Review namespace creation processes

Now that existing namespaces have been updated to enforce Pod Security Admission, you should ensure that your processes and/or policies for creating new namespaces are updated to ensure that an appropriate Pod Security profile is applied to new namespaces.

You can also statically configure the Pod Security admission controller to set a default enforce, audit, and/or warn level for unlabeled namespaces. See Configure the Admission Controller for more information.

5. Disable PodSecurityPolicy

Finally, you're ready to disable PodSecurityPolicy. To do so, you will need to modify the admission configuration of the API server: How do I turn off an admission controller?.

To verify that the PodSecurityPolicy admission controller is no longer enabled, you can manually run a test by impersonating a user without access to any PodSecurityPolicies (see the PodSecurityPolicy example), or by verifying in the API server logs. At startup, the API server outputs log lines listing the loaded admission controller plugins:

```
I0218 00:59:44.903329      13 plugins.go:158] Loaded 16 mutating
admission controller(s) successfully in the following order:
NamespaceLifecycle,LimitRanger,ServiceAccount,NodeRestriction,Tai
ntNodesByCondition,Priority,DefaultTolerationSeconds,ExtendedReso
urceToleration,PersistentVolumeLabel,DefaultStorageClass,StorageO
bjectInUseProtection,RuntimeClass,DefaultIngressClass,MutatingAdm
issionWebhook.
I0218 00:59:44.903350      13 plugins.go:161] Loaded 14
validating admission controller(s) successfully in the following
order:
LimitRanger,ServiceAccount,PodSecurity,Priority,PersistentVolumeC
laimResize,RuntimeClass,CertificateApproval,CertificateSigning,Ce
rtificateSubjectRestriction,DenyServiceExternalIPs,ValidatingAdmi
ssionWebhook,ResourceQuota.
```

You should see `PodSecurity` (in the validating admission controllers), and neither list should contain `PodSecurityPolicy`.

Once you are certain the PSP admission controller is disabled (and after sufficient soak time to be confident you won't need to roll back), you are free to delete your PodSecurityPolicies and any associated Roles, ClusterRoles, RoleBindings and ClusterRoleBindings (just make sure they don't grant any other unrelated permissions).

Manage Kubernetes Objects

Declarative and imperative paradigms for interacting with the Kubernetes API.

Declarative Management of Kubernetes Objects Using Configuration Files

Declarative Management of Kubernetes Objects Using Kustomize

Managing Kubernetes Objects Using Imperative Commands

Imperative Management of Kubernetes Objects Using Configuration Files

Update API Objects in Place Using kubectl patch

Use kubectl patch to update Kubernetes API objects in place. Do a strategic merge patch or a JSON merge patch.

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. `kubectl diff` also gives you a preview of what changes `apply` will make.

Before you begin

Install `kubectl`.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

Overview

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:

application/simple_deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Run `kubectl diff` to print the object that will be created:

```
kubectl diff -f https://k8s.io/examples/application/
simple_deployment.yaml
```

Note:

`diff` uses server-side dry-run, which needs to be enabled on `kube-apiserver`.

Since `diff` performs a server-side apply request in dry-run mode, it requires granting `PATCH`, `CREATE`, and `UPDATE` permissions. See Dry-Run Authorization for details.

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/
simple_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/
simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```yaml
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.14.2","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14.2
        # ...
        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 diff -f <directory>/
kubectl apply -f <directory>/
```

Note: Add the `-R` flag to recursively process directories.

Here's an example configuration file:

application/simple_deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/
simple_deployment.yaml
```

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/examples/application/
simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```yaml
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.14.2","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14.2
        # ...
        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 deployment nginx-deployment -o yaml
```

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/v1
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.14.2","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14.2
        # ...
        name: nginx
        ports:
        - containerPort: 80
      # ...
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.14.2` to `nginx:1.16.1`, and delete the `minReadySeconds` field:

application/update_deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1 # update the image
        ports:
        - containerPort: 80
```

Apply the changes made to the configuration file:

```
kubectl diff -f https://k8s.io/examples/application/
update_deployment.yaml
kubectl apply -f https://k8s.io/examples/application/
update_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/
update_deployment.yaml -o yaml
```

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:1.16.1` from `nginx:1.14.2`.
- 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.

```yaml
apiVersion: apps/v1
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.16.1","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl
apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.16.1 # 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 has a `last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory/> --prune -l <labels>
```

Warning: 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

Caution: 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:

application/update_deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1 # update the image
        ports:
        - containerPort: 80
```

Also, suppose this is the live configuration for the same Deployment object:

```yaml
apiVersion: apps/v1
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"minReadySeconds":5,"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.14.2","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
  # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14.2
```

```
        # ...
        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. Clear fields explicitly set to null in the local object configuration file regardless of whether they appear in the `last-applied-configuration`. 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/v1
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/v1","kind":"Deployment",
      "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
      "spec":{"selector":{"matchLabels":
{"app":nginx}},"template":{"metadata":{"labels":{"app":"nginx"}},
      "spec":{"containers":[{"image":"nginx:
1.16.1","name":"nginx",
      "ports":[{"containerPort":80}]}]}}}}
    # ...
spec:
  selector:
    matchLabels:
      # ...
      app: nginx
  replicas: 2 # Set by `kubectl scale`.  Ignored by `kubectl
```

```yaml
apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.16.1 # 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 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 if all its elements are primitives.
- 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 if all its elements are primitives

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`.

```
# last-applied-configuration value
    containers:
    - name: nginx
      image: nginx:1.16
    - 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:1.16
    - 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:1.16
    - 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:1.16
      # 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`:

application/simple_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/
simple_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/
simple_deployment.yaml -o yaml
```

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.

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
      app: nginx
```

```yaml
  minReadySeconds: 5
  replicas: 1 # defaulted by apiserver
  strategy:
    rollingUpdate: # defaulted by apiserver - derived from strategy.type
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate # defaulted by apiserver
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:1.14.2
        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
# ...
```

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:

```yaml
# last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
```

```yaml
        image: nginx:1.14.2
        ports:
        - containerPort: 80

# configuration file
spec:
  strategy:
    type: Recreate # updated value
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        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:1.14.2
        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:1.14.2
        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

Fields that do not appear in the configuration file can be cleared by setting their values to `null` and then applying the configuration file. For fields defaulted by the server, this triggers re-defaulting the values.

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.

Note: 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 > <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: "apps/v1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Kubectl Command Reference
- Kubernetes API Reference

Declarative Management of Kubernetes Objects Using Kustomize

Kustomize is a standalone tool to customize Kubernetes objects through a kustomization file.

Since 1.14, Kubectl also supports the management of Kubernetes objects using a kustomization file. To view Resources found in a directory containing a kustomization file, run the following command:

```
kubectl kustomize <kustomization_directory>
```

To apply those Resources, run `kubectl apply` with `--kustomize` or `-k` flag:

```
kubectl apply -k <kustomization_directory>
```

Before you begin

Install `kubectl`.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

Overview of Kustomize

Kustomize is a tool for customizing Kubernetes configurations. It has the following features to manage application configuration files:

- generating resources from other sources
- setting cross-cutting fields for resources
- composing and customizing collections of resources

Generating Resources

ConfigMaps and Secrets hold configuration or sensitive data that are used by other Kubernetes objects, such as Pods. The source of truth of ConfigMaps or Secrets are usually external to a cluster, such as a `.properties` file or an SSH keyfile. Kustomize has `secretGenerator` and `configMapGenerator`, which generate Secret and ConfigMap from files or literals.

configMapGenerator

To generate a ConfigMap from a file, add an entry to the `files` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a `.properties` file:

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
EOF
```

The generated ConfigMap can be examined with the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```yaml
apiVersion: v1
data:
  application.properties: |
        FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-8mbdf7882g
```

To generate a ConfigMap from an env file, add an entry to the `envs` list in `configMapGenerator`. This can also be used to set values from local environment variables by omitting the `=` and the value.

Note: It's recommended to use the local environment variable population functionality sparingly - an overlay with a patch is often more maintainable. Setting values from the environment may be useful when they cannot easily be predicted, such as a git SHA.

Here is an example of generating a ConfigMap with a data item from a `.env` file:

```shell
# Create a .env file
# BAZ will be populated from the local environment variable $BAZ
cat <<EOF >.env
FOO=Bar
BAZ
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  envs:
  - .env
EOF
```

The generated ConfigMap can be examined with the following command:

```shell
BAZ=Qux kubectl kustomize ./
```

The generated ConfigMap is:

```yaml
apiVersion: v1
data:
  BAZ: Qux
  FOO: Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-892ghb99c8
```

Note: Each variable in the `.env` file becomes a separate key in the ConfigMap that you generate. This is different from the previous example which embeds a file named `.properties` (and all its entries) as the value for a single key.

ConfigMaps can also be generated from literal key-value pairs. To generate a ConfigMap from a literal key-value pair, add an entry to the `literals` list in configMapGenerator. Here is an example of generating a ConfigMap with a data item from a key-value pair:

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-2
  literals:
  - FOO=Bar
EOF
```

The generated ConfigMap can be checked by the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  name: example-configmap-2-g2hdhfc6tk
```

To use a generated ConfigMap in a Deployment, reference it by the name of the configMapGenerator. Kustomize will automatically replace this name with the generated name.

This is an example deployment that uses a generated ConfigMap:

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
```

```
      - name: app
        image: my-app
        volumeMounts:
        - name: config
          mountPath: /config
      volumes:
      - name: config
        configMap:
          name: example-configmap-1
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
EOF
```

Generate the ConfigMap and Deployment:

```
kubectl kustomize ./
```

The generated Deployment will refer to the generated ConfigMap by name:

```
apiVersion: v1
data:
  application.properties: |
        FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-g4hk9g2ff8
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-app
  name: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - image: my-app
        name: app
        volumeMounts:
        - mountPath: /config
          name: config
      volumes:
      - configMap:
          name: example-configmap-1-g4hk9g2ff8
        name: config
```

secretGenerator

You can generate Secrets from files or literal key-value pairs. To generate a Secret from a file, add an entry to the `files` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a file:

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-1
  files:
  - password.txt
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:
  password.txt: dXNlcm5hbWU9YWRtaW4KcGFzc3dvcmQ9c2VjcmV0Cg==
kind: Secret
metadata:
  name: example-secret-1-t2kt65hgtb
type: Opaque
```

To generate a Secret from a literal key-value pair, add an entry to `literals` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a key-value pair:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-2
  literals:
  - username=admin
  - password=secret
EOF
```

The generated Secret is as follows:

```yaml
apiVersion: v1
data:
  password: c2VjcmV0
  username: YWRtaW4=
kind: Secret
metadata:
  name: example-secret-2-t52t6g96d8
type: Opaque
```

Like ConfigMaps, generated Secrets can be used in Deployments by referring to the name of the secretGenerator:

```shell
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: app
        image: my-app
        volumeMounts:
        - name: password
          mountPath: /secrets
      volumes:
      - name: password
        secret:
          secretName: example-secret-1
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
secretGenerator:
- name: example-secret-1
  files:
```

```
  - password.txt
EOF
```

generatorOptions

The generated ConfigMaps and Secrets have a content hash suffix appended. This ensures that a new ConfigMap or Secret is generated when the contents are changed. To disable the behavior of appending a suffix, one can use `generatorOptions`. Besides that, it is also possible to specify cross-cutting options for generated ConfigMaps and Secrets.

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-3
  literals:
  - FOO=Bar
generatorOptions:
  disableNameSuffixHash: true
  labels:
    type: generated
  annotations:
    note: generated
EOF
```

Run`kubectl kustomize ./` to view the generated ConfigMap:

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  annotations:
    note: generated
  labels:
    type: generated
  name: example-configmap-3
```

Setting cross-cutting fields

It is quite common to set cross-cutting fields for all Kubernetes resources in a project. Some use cases for setting cross-cutting fields:

- setting the same namespace for all Resources
- adding the same name prefix or suffix
- adding the same set of labels
- adding the same set of annotations

Here is an example:

```
# Create a deployment.yaml
cat <<EOF >./deployment.yaml
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
EOF

cat <<EOF >./kustomization.yaml
namespace: my-namespace
namePrefix: dev-
nameSuffix: "-001"
commonLabels:
  app: bingo
commonAnnotations:
  oncallPager: 800-555-1212
resources:
- deployment.yaml
EOF
```

Run `kubectl kustomize ./` to view those fields are all set in the Deployment Resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    oncallPager: 800-555-1212
  labels:
    app: bingo
  name: dev-nginx-deployment-001
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: bingo
  template:
    metadata:
      annotations:
        oncallPager: 800-555-1212
      labels:
        app: bingo
    spec:
      containers:
      - image: nginx
        name: nginx
```

Composing and Customizing Resources

It is common to compose a set of Resources in a project and manage them inside the same file or directory. Kustomize offers composing Resources from different files and applying patches or other customization to them.

Composing

Kustomize supports composition of different resources. The `resources` field, in the `kustomization.yaml` file, defines the list of resources to include in a configuration. Set the path to a resource's configuration file in the `resources` list. Here is an example of an NGINX application comprised of a Deployment and a Service:

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
```

```
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF

# Create a kustomization.yaml composing them
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

The Resources from `kubectl kustomize ./` contain both the Deployment and the Service objects.

Customizing

Patches can be used to apply different customizations to Resources. Kustomize supports different patching mechanisms through `patchesStrategicMerge` and `patchesJson6902`. `patchesStrategicMerge` is a list of file paths. Each file should be resolved to a strategic merge patch. The names inside the patches must match Resource names that are already loaded. Small patches that do one thing are recommended. For example, create one patch for increasing the deployment replica number and another patch for setting the memory limit.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF
```

```
# Create a patch increase_replicas.yaml
cat <<EOF > increase_replicas.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
EOF

# Create another patch set_memory.yaml
cat <<EOF > set_memory.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
      - name: my-nginx
        resources:
          limits:
            memory: 512Mi
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
patchesStrategicMerge:
- increase_replicas.yaml
- set_memory.yaml
EOF
```

Run `kubectl kustomize ./` to view the Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - image: nginx
```

```
        name: my-nginx
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: 512Mi
```

Not all Resources or fields support strategic merge patches. To support modifying arbitrary fields in arbitrary Resources, Kustomize offers applying JSON patch through `patchesJson6902`. To find the correct Resource for a Json patch, the group, version, kind and name of that Resource need to be specified in `kustomization.yaml`. For example, increasing the replica number of a Deployment object can also be done through `patchesJson6902`.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a json patch
cat <<EOF > patch.yaml
- op: replace
  path: /spec/replicas
  value: 3
EOF

# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml

patchesJson6902:
- target:
    group: apps
    version: v1
    kind: Deployment
    name: my-nginx
  path: patch.yaml
EOF
```

Run `kubectl kustomize ./` to see the `replicas` field is updated:

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

In addition to patches, Kustomize also offers customizing container images or injecting field values from other objects into containers without creating patches. For example, you can change the image used inside containers by specifying the new image in `images` field in `kustomization.yaml`.

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
```

```
        - containerPort: 80
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
images:
- name: nginx
  newName: my.image.registry/nginx
  newTag: 1.4.0
EOF
```

Run `kubectl kustomize ./` to see that the image being used is updated:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - image: my.image.registry/nginx:1.4.0
        name: my-nginx
        ports:
        - containerPort: 80
```

Sometimes, the application running in a Pod may need to use configuration values from other objects. For example, a Pod from a Deployment object need to read the corresponding Service name from Env or as a command argument. Since the Service name may change as `namePrefix` or `nameSuffix` is added in the `kustomization.yaml` file. It is not recommended to hard code the Service name in the command argument. For this usage, Kustomize can inject the Service name into containers through `vars`.

```
# Create a deployment.yaml file (quoting the here doc delimiter)
cat <<'EOF' > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        command: ["start", "--host", "$(MY_SERVICE_NAME)"]
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF

cat <<EOF >./kustomization.yaml
namePrefix: dev-
nameSuffix: "-001"

resources:
- deployment.yaml
- service.yaml

vars:
- name: MY_SERVICE_NAME
  objref:
    kind: Service
    name: my-nginx
    apiVersion: v1
EOF
```

Run `kubectl kustomize ./` to see that the Service name injected into containers is `dev-my-nginx-001`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dev-my-nginx-001
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - command:
        - start
        - --host
        - dev-my-nginx-001
        image: nginx
        name: my-nginx
```

Bases and Overlays

Kustomize has the concepts of **bases** and **overlays**. A **base** is a directory with a `kustomization.yaml`, which contains a set of resources and associated customization. A base could be either a local directory or a directory from a remote repo, as long as a `kustomization.yaml` is present inside. An **overlay** is a directory with a `kustomization.yaml` that refers to other kustomization directories as its `bases`. A **base** has no knowledge of an overlay and can be used in multiple overlays. An overlay may have multiple bases and it composes all resources from bases and may also have customization on top of them.

Here is an example of a base:

```
# Create a directory to hold the base
mkdir base
# Create a base/deployment.yaml
cat <<EOF > base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
```

```
        image: nginx
EOF

# Create a base/service.yaml file
cat <<EOF > base/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF
# Create a base/kustomization.yaml
cat <<EOF > base/kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

This base can be used in multiple overlays. You can add different `namePrefix` or other cross-cutting fields in different overlays. Here are two overlays using the same base.

```
mkdir dev
cat <<EOF > dev/kustomization.yaml
bases:
- ../base
namePrefix: dev-
EOF

mkdir prod
cat <<EOF > prod/kustomization.yaml
bases:
- ../base
namePrefix: prod-
EOF
```

How to apply/view/delete objects using Kustomize

Use `--kustomize` or `-k` in `kubectl` commands to recognize Resources managed by `kustomization.yaml`. Note that `-k` should point to a kustomization directory, such as

```
kubectl apply -k <kustomization directory>/
```

Given the following `kustomization.yaml`,

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
EOF

# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
namePrefix: dev-
commonLabels:
  app: my-nginx
resources:
- deployment.yaml
EOF
```

Run the following command to apply the Deployment object `dev-my-nginx`:

```
> kubectl apply -k ./
deployment.apps/dev-my-nginx created
```

Run one of the following commands to view the Deployment object `dev-my-nginx`:

```
kubectl get -k ./
```

```
kubectl describe -k ./
```

Run the following command to compare the Deployment object `dev-my-nginx` against the state that the cluster would be in if the manifest was applied:

```
kubectl diff -k ./
```

Run the following command to delete the Deployment object `dev-my-nginx`:

```
> kubectl delete -k ./
deployment.apps "dev-my-nginx" deleted
```

Kustomize Feature List

Field	**Type**	**Explanation**
namespace	string	add namespace to all resources
namePrefix	string	value of this field is prepended to the names of all resources
nameSuffix	string	value of this field is appended to the names of all resources
commonLabels	map[string]string	labels to add to all resources and selectors
commonAnnotations	map[string]string	annotations to add to all resources
resources	[]string	each entry in this list must resolve to an existing resource configuration file
configMapGenerator	[]ConfigMapArgs	Each entry in this list generates a ConfigMap
secretGenerator	[]SecretArgs	Each entry in this list generates a Secret
generatorOptions	GeneratorOptions	Modify behaviors of all ConfigMap and Secret generator
bases	[]string	Each entry in this list should resolve to a directory containing a kustomization.yaml file
patchesStrategicMerge	[]string	Each entry in this list should resolve a strategic merge patch of a Kubernetes object
patchesJson6902	[]Patch	Each entry in this list should resolve to a Kubernetes object and a Json Patch
vars	[]Var	Each entry is to capture text from one resource's field
images	[]Image	Each entry is to modify the name, tags and/or digest for one image without creating patches
configurations	[]string	Each entry in this list should resolve to a file containing Kustomize transformer configurations
crds	[]string	Each entry in this list should resolve to an OpenAPI definition file for Kubernetes types

What's next

- Kustomize
- Kubectl Book
- Kubectl Command Reference
- Kubernetes API 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.

Before you begin

Install `kubectl`.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 Pod to run a Container.

- `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` `<field>`: 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 my-svc --clusterip="None" -o yaml --dry-run=client | kubectl set selector --local -f - 'environment=qa' -o yaml | kubectl create -f -
```

1. The `kubectl create service -o yaml --dry-run=client` 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 selector --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 --clusterip="None" -o
yaml --dry-run=client > /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 API 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.

Before you begin

Install `kubectl`.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 API 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>`

Note:

If configuration file has specified the `generateName` field in the `metadata` section instead of the `name` field, you cannot delete the object using `kubectl delete -f <filename|url>`. You will have to use other flags for deleting the object. For example:

```
kubectl delete <type> <name>
kubectl delete <type> -l <label>
```

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 > <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:

```yaml
selector:
  matchLabels:
      controller-selector: "apps/v1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "apps/v1/deployment/nginx"
```

What's next

- Managing Kubernetes Objects Using Imperative Commands
- Managing Kubernetes Objects Using Object Configuration (Declarative)
- Kubectl Command Reference
- Kubernetes API Reference

Update API Objects in Place Using kubectl patch

Use kubectl patch to update Kubernetes API objects in place. Do a strategic merge patch or a JSON merge 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

application/deployment-patch.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patch-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: patch-demo-ctr
        image: nginx
      tolerations:
      - effect: NoSchedule
        key: dedicated
        value: test-team
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-patch.yaml
```

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:

```yaml
spec:
  template:
    spec:
      containers:
      - name: patch-demo-ctr-2
        image: redis
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch-file 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

The patch you did in the preceding exercise is called a *strategic merge patch*. Notice that the patch did not replace the `containers` list. Instead it added a new Container to the list. In other words, the list in the patch was merged with the existing list. This is not always what happens when you use a strategic merge patch on a list. In some cases, the list is replaced, not merged.

With a strategic merge patch, a list is either replaced or merged depending on its patch strategy. The patch strategy is specified by the value of the `patchStrategy` key in a field tag in the Kubernetes source code. For example, the `Containers` field of `PodSpec` struct has a `patchStrategy` of `merge`:

```
type PodSpec struct {
  ...
  Containers []Container `json:"containers"
patchStrategy:"merge" patchMergeKey:"name" ...`
```

You can also see the patch strategy in the OpenApi spec:

```
"io.k8s.api.core.v1.PodSpec": {
    ...
     "containers": {
      "description": "List of containers belonging to the
pod. ...
      },
      "x-kubernetes-patch-merge-key": "name",
      "x-kubernetes-patch-strategy": "merge"
     },
```

And you can see the patch strategy in the Kubernetes API documentation.

Create a file named `patch-file-tolerations.yaml` that has this content:

```
spec:
  template:
    spec:
      tolerations:
      - effect: NoSchedule
        key: disktype
        value: ssd
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch-file patch-file-
tolerations.yaml
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has only one Toleration:

```yaml
tolerations:
- effect: NoSchedule
  key: disktype
  value: ssd
```

Notice that the `tolerations` list in the PodSpec was replaced, not merged. This is because the Tolerations field of PodSpec does not have a `patchStrategy` key in its field tag. So the strategic merge patch uses the default patch strategy, which is `replace`.

```go
type PodSpec struct {
  ...
  Tolerations []Toleration `json:"tolerations,omitempty"
protobuf:"bytes,22,opt,name=tolerations"`
```

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:

```yaml
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-file
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
```

Use strategic merge patch to update a Deployment using the retainKeys strategy

Here's the configuration file for a Deployment that uses the `RollingUpdate` strategy:

application/deployment-retainkeys.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: retainkeys-demo
spec:
  selector:
    matchLabels:
      app: nginx
```

```yaml
  strategy:
    rollingUpdate:
      maxSurge: 30%
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: retainkeys-demo-ctr
        image: nginx
```

Create the deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-retainkeys.yaml
```

At this point, the deployment is created and is using the `RollingUpdate` strategy.

Create a file named `patch-file-no-retainkeys.yaml` that has this content:

```yaml
spec:
  strategy:
    type: Recreate
```

Patch your Deployment:

```
kubectl patch deployment retainkeys-demo --type merge --patch-file patch-file-no-retainkeys.yaml
```

In the output, you can see that it is not possible to set `type` as `Recreate` when a value is defined for `spec.strategy.rollingUpdate`:

```
The Deployment "retainkeys-demo" is invalid:
spec.strategy.rollingUpdate: Forbidden: may not be specified
when strategy `type` is 'Recreate'
```

The way to remove the value for `spec.strategy.rollingUpdate` when updating the value for `type` is to use the `retainKeys` strategy for the strategic merge.

Create another file named `patch-file-retainkeys.yaml` that has this content:

```yaml
spec:
  strategy:
    $retainKeys:
    - type
    type: Recreate
```

With this patch, we indicate that we want to retain only the `type` key of the `strategy` object. Thus, the `rollingUpdate` will be removed during the patch operation.

Patch your Deployment again with this new patch:

```
kubectl patch deployment retainkeys-demo --type merge --patch-
file patch-file-retainkeys.yaml
```

Examine the content of the Deployment:

```
kubectl get deployment retainkeys-demo --output yaml
```

The output shows that the strategy object in the Deployment does not contain the `rollingUpdate` key anymore:

```
spec:
  strategy:
    type: Recreate
  template:
```

Notes on the strategic merge patch using the retainKeys strategy

The patch you did in the preceding exercise is called a *strategic merge patch with retainKeys strategy*. This method introduces a new directive `$retainKeys` that has the following strategies:

- It contains a list of strings.
- All fields needing to be preserved must be present in the `$retainKeys` list.
- The fields that are present will be merged with live object.
- All of the missing fields will be cleared when patching.
- All fields in the `$retainKeys` list must be a superset or the same as the fields present in the patch.

The `retainKeys` strategy does not work for all objects. It only works when the value of the `patchStrategy` key in a field tag in the Kubernetes source code contains `retainKeys`. For example, the `Strategy` field of the `DeploymentSpec` struct has a `patchStrategy` of `retainKeys`:

```
type DeploymentSpec struct {
  ...
  // +patchStrategy=retainKeys
  Strategy DeploymentStrategy `json:"strategy,omitempty"
patchStrategy:"retainKeys" ...`
```

You can also see the `retainKeys` strategy in the OpenApi spec:

```
"io.k8s.api.apps.v1.DeploymentSpec": {
   ...
  "strategy": {
    "$ref": "#/definitions/
io.k8s.api.apps.v1.DeploymentStrategy",
    "description": "The deployment strategy to use to replace
existing pods with new ones.",
    "x-kubernetes-patch-strategy": "retainKeys"
  },
```

And you can see the `retainKeys` strategy in the Kubernetes API documentation.

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-file patch-file.yaml
kubectl patch deployment patch-demo --patch 'spec:\n template:
\n  spec:\n   containers:\n   - name: patch-demo-ctr-2\n
image: redis'

kubectl patch deployment patch-demo --patch-file patch-file.json
kubectl patch deployment patch-demo --patch '{"spec":
{"template": {"spec": {"containers": [{"name": "patch-demo-
ctr-2","image": "redis"}]}}}}'
```

Summary

In this exercise, you used `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, and kubectl apply.

Note: Strategic merge patch is not supported for custom resources.

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

Managing Secrets

Managing confidential settings data using Secrets.

Managing Secrets using kubectl

Creating Secret objects using kubectl command line.

Managing Secrets using Configuration File

Creating Secret objects using resource configuration file.

Managing Secrets using Kustomize

Creating Secret objects using kustomization.yaml file.

Managing Secrets using kubectl

Creating Secret objects using kubectl command line.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 Secret

A `Secret` can contain user credentials required by pods to access a database. For example, a database connection string consists of a username and password. You can store the username in a file `./username.txt` and the password in a file `./password.txt` on your local machine.

```
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt
```

In these commands, the `-n` flag ensures that the generated files do not have an extra newline character at the end of the text. This is important because when `kubectl` reads a file and encodes the content into a base64 string, the extra newline character gets encoded too.

The `kubectl create secret` command packages these files into a Secret and creates the object on the API server.

```
kubectl create secret generic db-user-pass \
  --from-file=./username.txt \
  --from-file=./password.txt
```

The output is similar to:

```
secret/db-user-pass created
```

The default key name is the filename. You can optionally set the key name using `--from-file=[key=]source`. For example:

```
kubectl create secret generic db-user-pass \
  --from-file=username=./username.txt \
  --from-file=password=./password.txt
```

You do not need to escape special characters in password strings that you include in a file.

You can also provide Secret data using the `--from-literal=<key>=<value>` tag. This tag can be specified more than once to provide multiple key-value pairs. Note that special characters such as `$`, `\`, `*`, `=`, and `!` will be interpreted by your shell and require escaping.

In most shells, the easiest way to escape the password is to surround it with single quotes (`'`). For example, if your password is `S!B\*d$zDsb=`, run the following command:

```
kubectl create secret generic db-user-pass \
  --from-literal=username=devuser \
  --from-literal=password='S!B\*d$zDsb='
```

Verify the Secret

Check that the Secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME                      TYPE
DATA      AGE
```

```
db-user-pass          Opaque
2         51s
```

You can view a description of the `Secret`:

```
kubectl describe secrets/db-user-pass
```

The output is similar to:

```
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password:    12 bytes
username:    5 bytes
```

The commands `kubectl get` and `kubectl describe` avoid showing the contents of a `Secret` by default. This is to protect the `Secret` from being exposed accidentally, or from being stored in a terminal log.

Decoding the Secret

To view the contents of the Secret you created, run the following command:

```
kubectl get secret db-user-pass -o jsonpath='{.data}'
```

The output is similar to:

```
{"password":"MWYyZDFlMmU2N2Rm","username":"YWRtaW4="}
```

Now you can decode the `password` data:

```
# This is an example for documentation purposes.
# If you did things this way, the data 'MWYyZDFlMmU2N2Rm' could
be stored in
# your shell history.
# Someone with access to you computer could find that remembered
command
# and base-64 decode the secret, perhaps without your knowledge.
# It's usually better to combine the steps, as shown later in
the page.
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

The output is similar to:

```
1f2d1e2e67df
```

In order to avoid storing a secret encoded value in your shell history, you can run the following command:

```
kubectl get secret db-user-pass -o jsonpath='{.data.password}' |
base64 --decode
```

The output shall be similar as above.

Clean Up

Delete the Secret you created:

```
kubectl delete secret db-user-pass
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets using config files
- Learn how to manage Secrets using kustomize

Managing Secrets using Configuration File

Creating Secret objects using resource configuration file.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 the Config file

You can create a Secret in a file first, in JSON or YAML format, and then create that object. The Secret resource contains two maps: `data` and `stringData`. The `data` field is used to store arbitrary data, encoded using base64. The `stringData` field is provided for convenience, and it allows you to provide Secret data as unencoded strings. The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`.

For example, to store two strings in a Secret using the `data` field, convert the strings to base64 as follows:

```
echo -n 'admin' | base64
```

The output is similar to:

```
YWRtaW4=
```

```
echo -n '1f2d1e2e67df' | base64
```

The output is similar to:

```
MWYyZDFlMmU2N2Rm
```

Write a Secret config file that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Note that the name of a Secret object must be a valid DNS subdomain name.

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/macOS, 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 the `-w` option is not available.

For certain scenarios, you may wish to use the `stringData` field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

For example, if your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "<user>"
password: "<password>"
```

You could store this in a Secret using the following definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
```

```
stringData:
  config.yaml: |
    apiUrl: "https://my.api.com/api/v1"
    username: <user>
    password: <password>
```

Create the Secret object

Now create the Secret using kubectl apply:

```
kubectl apply -f ./secret.yaml
```

The output is similar to:

```
secret/mysecret created
```

Check the Secret

The `stringData` field is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdX
Nlcm5hbWU6IHt7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
```

The commands `kubectl get` and `kubectl describe` avoid showing the contents of a `Secret` by default. This is to protect the `Secret` from being exposed accidentally to an onlooker, or from being stored in a terminal log. To check the actual content of the encoded data, please refer to decoding secret.

If a field, such as `username`, is specified in both `data` and `stringData`, the value from `stringData` is used. For example, the following Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
```

```
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Results in the following Secret:

```
apiVersion: v1
data:
  username: YWRtaW5pc3RyYXRvcg==
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
```

Where `YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

Clean Up

To delete the Secret you have created:

```
kubectl delete secret mysecret
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets with the `kubectl` command
- Learn how to manage Secrets using kustomize

Managing Secrets using Kustomize

Creating Secret objects using kustomization.yaml file.

Since Kubernetes v1.14, `kubectl` supports managing objects using Kustomize. Kustomize provides resource Generators to create Secrets and ConfigMaps. The Kustomize generators should be specified in a `kustomization.yaml` file inside a directory. After generating the Secret, you can create the Secret on the API server with `kubectl apply`.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to

run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 the Kustomization file

You can generate a Secret by defining a `secretGenerator` in a `kustomization.yaml` file that references other existing files. For example, the following kustomization file references the `./username.txt` and the `./password.txt` files:

```
secretGenerator:
- name: db-user-pass
  files:
  - username.txt
  - password.txt
```

You can also define the `secretGenerator` in the `kustomization.yaml` file by providing some literals. For example, the following `kustomization.yaml` file contains two literals for `username` and `password` respectively:

```
secretGenerator:
- name: db-user-pass
  literals:
  - username=admin
  - password=1f2d1e2e67df
```

You can also define the `secretGenerator` in the `kustomization.yaml` file by providing `.env` files. For example, the following `kustomization.yaml` file pulls in data from `.env.secret` file:

```
secretGenerator:
- name: db-user-pass
  envs:
  - .env.secret
```

Note that in all cases, you don't need to base64 encode the values.

Create the Secret

Apply the directory containing the `kustomization.yaml` to create the Secret.

```
kubectl apply -k .
```

The output is similar to:

```
secret/db-user-pass-96mffmfh4k created
```

Note that when a Secret is generated, the Secret name is created by hashing the Secret data and appending the hash value to the name. This ensures that a new Secret is generated each time the data is modified.

Check the Secret created

You can check that the secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME
TYPE                                      DATA      AGE
db-user-pass-96mffmfh4k
Opaque                                    2         51s
```

You can view a description of the secret:

```
kubectl describe secrets/db-user-pass-96mffmfh4k
```

The output is similar to:

```
Name:            db-user-pass-96mffmfh4k
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

The commands `kubectl get` and `kubectl describe` avoid showing the contents of a `Secret` by default. This is to protect the `Secret` from being exposed accidentally to an onlooker, or from being stored in a terminal log. To check the actual content of the encoded data, please refer to decoding secret.

Clean Up

To delete the Secret you have created:

```
kubectl delete secret db-user-pass-96mffmfh4k
```

What's next

- Read more about the Secret concept
- Learn how to manage Secrets with the `kubectl` command

- Learn how to manage Secrets using config file

Inject Data Into Applications

Specify configuration and other data for the Pods that run your workload.

Define a Command and Arguments for a Container

Define Dependent Environment Variables

Define Environment Variables for a Container

Expose Pod Information to Containers Through Environment Variables

Expose Pod Information to Containers Through Files

Distribute Credentials Securely Using Secrets

Define a Command and Arguments for a Container

This page shows how to define commands and arguments when you run a container in a 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

Note: The `command` field corresponds to `entrypoint` in some container runtimes.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a command and two arguments:

pods/commands.yaml

```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"]
  restartPolicy: OnFailure
```

1. Create a Pod based on the YAML configuration file:

   ```
   kubectl apply -f https://k8s.io/examples/pods/commands.yaml
   ```

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"]
```

What's next

- Learn more about configuring pods and containers.
- Learn more about running commands in a container.
- See Container.

Define Dependent Environment Variables

This page shows how to define dependent environment variables for 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as

control plane hosts. 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 dependent variable for a container

When you create a Pod, you can set dependent environment variables for the containers that run in the Pod. To set dependent environment variables, you can use $(VAR_NAME) in the `value` of `env` in the configuration file.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines an dependent environment variable with common usage defined. Here is the configuration manifest for the Pod:

pods/inject/dependent-envars.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dependent-envars-demo
spec:
  containers:
    - name: dependent-envars-demo
      args:
        - while true; do echo -en '\n'; printf
UNCHANGED_REFERENCE=$UNCHANGED_REFERENCE'\n'; printf
SERVICE_ADDRESS=$SERVICE_ADDRESS'\n';printf
ESCAPED_REFERENCE=$ESCAPED_REFERENCE'\n'; sleep 30; done;
      command:
        - sh
        - -c
      image: busybox:1.28
      env:
        - name: SERVICE_PORT
          value: "80"
        - name: SERVICE_IP
          value: "172.17.0.1"
        - name: UNCHANGED_REFERENCE
          value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
        - name: PROTOCOL
          value: "https"
        - name: SERVICE_ADDRESS
          value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
        - name: ESCAPED_REFERENCE
          value: "$$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
```

1. Create a Pod based on that manifest:

```
kubectl apply -f https://k8s.io/examples/pods/inject/
dependent-envars.yaml
```

```
pod/dependent-envars-demo created
```

2. List the running Pods:

```
kubectl get pods dependent-envars-demo
```

```
NAME                      READY     STATUS    RESTARTS   AGE
dependent-envars-demo     1/1       Running   0          9s
```

3. Check the logs for the container running in your Pod:

```
kubectl logs pod/dependent-envars-demo
```

```

UNCHANGED_REFERENCE=$(PROTOCOL)://172.17.0.1:80
SERVICE_ADDRESS=https://172.17.0.1:80
ESCAPED_REFERENCE=$(PROTOCOL)://172.17.0.1:80
```

As shown above, you have defined the correct dependency reference of `SERVICE_ADDRESS`, bad dependency reference of `UNCHANGED_REFERENCE` and skip dependent references of `ESCAPED_REFERENCE`.

When an environment variable is already defined when being referenced, the reference can be correctly resolved, such as in the `SERVICE_ADDRESS` case.

When the environment variable is undefined or only includes some variables, the undefined environment variable is treated as a normal string, such as `UNCHANGED_REFERENCE`. Note that incorrectly parsed environment variables, in general, will not block the container from starting.

The `$(VAR_NAME)` syntax can be escaped with a double `$`, ie: `$$(VAR_NAME)`. Escaped references are never expanded, regardless of whether the referenced variable is defined or not. This can be seen from the `ESCAPED_REFERENCE` case above.

What's next

- Learn more about environment variables.
- See EnvVarSource.

Define Environment Variables for a Container

This page shows how to define environment variables for 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 manifest for the Pod:

pods/inject/envars.yaml

```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"
    - name: DEMO_FAREWELL
      value: "Such a sweet sorrow"
```

1. Create a Pod based on that manifest:

   ```
   kubectl apply -f https://k8s.io/examples/pods/inject/envars.yaml
   ```

2. List the running Pods:

   ```
   kubectl get pods -l purpose=demonstrate-envars
   ```

The output is similar to:

```
NAME           READY    STATUS    RESTARTS   AGE
envar-demo     1/1      Running   0          9s
```

3. List the Pod's container environment variables:

```
kubectl exec 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
DEMO_FAREWELL=Such a sweet sorrow
```

Note: The environment variables set using the `env` or `envFrom` field override any environment variables specified in the container image.
Note: Environment variables may reference each other, however ordering is important. Variables making use of others defined in the same context must come later in the list. Similarly, avoid circular references.

Using environment variables inside of your config

Environment variables that you define in a Pod's configuration can be used elsewhere in the configuration, for example in commands and arguments that you set for the Pod's containers. In the example configuration below, the `GREETING`, `HONORIFIC`, and `NAME` environment variables are set to `Warm greetings to`, `The Most Honorable`, and `Kubernetes`, respectively. Those environment variables are then used in the CLI arguments passed to the `env-print-demo` container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
  - name: env-print-demo
    image: bash
    env:
    - name: GREETING
      value: "Warm greetings to"
    - name: HONORIFIC
      value: "The Most Honorable"
    - name: NAME
      value: "Kubernetes"
    command: ["echo"]
    args: ["$(GREETING) $(HONORIFIC) $(NAME)"]
```

Upon creation, the command `echo Warm greetings to The Most Honorable Kubernetes` is run on the container.

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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- Volume Files

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:

pods/inject/dapi-envars-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/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 apply -f https://k8s.io/examples/pods/inject/dapi-envars-pod.yaml
```

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:

pods/inject/dapi-envars-container.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/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 apply -f https://k8s.io/examples/pods/inject/dapi-envars-container.yaml
```

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- Volume files

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:

`pods/inject/dapi-volume.yaml`

```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: k8s.gcr.io/busybox
      command: ["sh", "-c"]
      args:
      - while true; do
          if [[ -e /etc/podinfo/labels ]]; then
            echo -en '\n\n'; cat /etc/podinfo/labels; fi;
          if [[ -e /etc/podinfo/annotations ]]; then
            echo -en '\n\n'; cat /etc/podinfo/annotations; fi;
          sleep 5;
        done;
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
  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/podinfo`.

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 apply -f https://k8s.io/examples/pods/inject/dapi-
volume.yaml
```

Verify that the 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/podinfo/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/podinfo/annotations
```

View the files in the `/etc/podinfo` directory:

```
/# ls -laR /etc/podinfo
```

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/podinfo` directory, `..data` is a symbolic link to the temporary subdirectory. Also in the `/etc/podinfo` 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).

Note: A container using Downward API as a subPath volume mount will not receive Downward API updates.

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:

`pods/inject/dapi-volume-resources.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox:1.24
      command: ["sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          if [[ -e /etc/podinfo/cpu_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_limit; fi;
          if [[ -e /etc/podinfo/cpu_request ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_request; fi;
          if [[ -e /etc/podinfo/mem_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_limit; fi;
          if [[ -e /etc/podinfo/mem_request ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_request; fi;
          sleep 5;
        done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
              divisor: 1m
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
              divisor: 1m
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
              divisor: 1Mi
          - path: "mem_request"
```

```
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
              divisor: 1Mi
```

In the configuration file, you can see that the Pod has a [downwardAPI volume](), and the Container mounts the volume at `/etc/podinfo`.

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 in the format specified by `1m` should be stored in a file named `cpu_limit`. The `divisor` field is optional and has the default value of `1` which means cores for cpu and bytes for memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume-resources.yaml
```

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/podinfo/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 `downwardAPI` volumes:

- Information available via `fieldRef`:
    - `metadata.name` - the pod's name
    - `metadata.namespace` - the pod's namespace
    - `metadata.uid` - the pod's UID
    - `metadata.labels['<KEY>']` - the value of the pod's label `<KEY>` (for example, `metadata.labels['mylabel']`)
    - `metadata.annotations['<KEY>']` - the value of the pod's annotation `<KEY>` (for example, `metadata.annotations['myannotation']`)
- Information available via `resourceFieldRef`:
    - A Container's CPU limit
    - A Container's CPU request

    - A Container's memory limit
    - A Container's memory request
    - A Container's hugepages limit (provided that the `DownwardAPIHugePages` feature gate is enabled)
    - A Container's hugepages request (provided that the `DownwardAPIHugePages` feature gate is enabled)
    - A Container's ephemeral-storage limit
    - A Container's ephemeral-storage request

In addition, the following information is available through `downwardAPI` volume `fieldRef`:

- `metadata.labels` - all of the pod's labels, formatted as `label-key="escaped-label-value"` with one label per line
- `metadata.annotations` - all of the pod's annotations, formatted as `annotation-key="escaped-annotation-value"` with one annotation per line

The following information is available through environment variables:

- `status.podIP` - the pod's IP address
- `spec.serviceAccountName` - the pod's service account name
- `spec.nodeName` - the name of the node to which the scheduler always attempts to schedule the pod
- `status.hostIP` - the IP of the node to which the Pod is assigned

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

- Check the [PodSpec](#) API definition which defines the desired state of a Pod.
- Check the [`Volume`](#) API definition which defines a generic volume in a Pod for containers to access.
- Check the [`DownwardAPIVolumeSource`](#) API definition which defines a volume that contains Downward API information.
- Check the [`DownwardAPIVolumeFile`](#) API definition which contains references to object or resource fields for populating a file in the Downward API volume.
- Check the [`ResourceFieldSelector`](#) API definition which specifies the container resources and their output format.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 a base64 encoding tool to convert your username and password to a base64 representation. Here's an example using the commonly available base64 program:

```
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`.

Caution: Use a local tool trusted by your OS to decrease the security risks of external tools.

Create a Secret

Here is a configuration file you can use to create a Secret that holds your username and password:

pods/inject/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

1. Create the Secret

   ```
   kubectl apply -f https://k8s.io/examples/pods/inject/
   secret.yaml
   ```

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 Secret directly with kubectl

If you want to skip the Base64 encoding step, you can create the same Secret using the `kubectl create secret` command. For example:

```
kubectl create secret generic test-secret --from-literal='usernam
e=my-app' --from-literal='password=39528$vdg7Jb'
```

This is more convenient. The detailed approach shown earlier runs through each step explicitly to demonstrate what is happening.

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:

pods/inject/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 apply -f https://k8s.io/examples/pods/inject/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 -i -t 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, list the files in the `/etc/secret-volume` directory:

   ```
   # Run this in the shell inside the container
   ls /etc/secret-volume
   ```

   The output shows two files, one for each piece of secret data:

   ```
   password username
   ```

5. In your shell, display the contents of the `username` and `password` files:

   ```
   # Run this in the shell inside the container
   echo "$( cat /etc/secret-volume/username )"
   echo "$( cat /etc/secret-volume/password )"
   ```

   The output is your username and password:

   ```
   my-app
   39528$vdg7Jb
   ```

Define container environment variables using Secret data

Define a container environment variable with data from a single Secret

- Define an environment variable as a key-value pair in a Secret:

  ```
  kubectl create secret generic backend-user --from-literal=bac
  kend-username='backend-admin'
  ```

- Assign the `backend-username` value defined in the Secret to the `SECRET_USERNAME` environment variable in the Pod specification.

  pods/inject/pod-single-secret-env-variable.yaml

  ```
  apiVersion: v1
  kind: Pod
  metadata:
    name: env-single-secret
  spec:
    containers:
    - name: envars-test-container
      image: nginx
      env:
      - name: SECRET_USERNAME
        valueFrom:
  ```

```
        secretKeyRef:
          name: backend-user
          key: backend-username
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-single-secret-env-variable.yaml
```

- In your shell, display the content of `SECRET_USERNAME` container environment variable

```
kubectl exec -i -t env-single-secret -- /bin/sh -c 'echo $SECRET_USERNAME'
```

The output is

```
backend-admin
```

Define container environment variables with data from multiple Secrets

- As with the previous example, create the Secrets first.

```
kubectl create secret generic backend-user --from-literal=backend-username='backend-admin'
kubectl create secret generic db-user --from-literal=db-username='db-admin'
```

- Define the environment variables in the Pod specification.

pods/inject/pod-multiple-secret-env-variable.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: envvars-multiple-secrets
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: BACKEND_USERNAME
      valueFrom:
        secretKeyRef:
          name: backend-user
          key: backend-username
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-user
          key: db-username
```

- Create the Pod:

  ```
  kubectl create -f https://k8s.io/examples/pods/inject/pod-
  multiple-secret-env-variable.yaml
  ```

- In your shell, display the container environment variables

  ```
  kubectl exec -i -t envvars-multiple-secrets -- /bin/sh -c 'en
  v | grep _USERNAME'
  ```

  The output is

  ```
  DB_USERNAME=db-admin
  BACKEND_USERNAME=backend-admin
  ```

Configure all key-value pairs in a Secret as container environment variables

Note: This functionality is available in Kubernetes v1.6 and later.

- Create a Secret containing multiple key-value pairs

  ```
  kubectl create secret generic test-secret --from-literal=user
  name='my-app' --from-literal=password='39528$vdg7Jb'
  ```

- Use envFrom to define all of the Secret's data as container environment variables. The key from the Secret becomes the environment variable name in the Pod.

  pods/inject/pod-secret-envFrom.yaml

  ```
  apiVersion: v1
  kind: Pod
  metadata:
    name: envfrom-secret
  spec:
    containers:
    - name: envars-test-container
      image: nginx
      envFrom:
      - secretRef:
          name: test-secret
  ```

- Create the Pod:

  ```
  kubectl create -f https://k8s.io/examples/pods/inject/pod-
  secret-envFrom.yaml
  ```

- In your shell, display `username` and `password` container environment variables

  ```
  kubectl exec -i -t envfrom-secret -- /bin/sh -c 'echo
  "username: $username\npassword: $password\n"'
  ```

  The output is

  ```
  username: my-app
  password: 39528$vdg7Jb
  ```

References

- Secret
- Volume
- Pod

What's next

- Learn more about Secrets.
- Learn about Volumes.

Run Applications

Run and manage both stateless and stateful applications.

Run a Stateless Application Using a Deployment

Run a Single-Instance Stateful Application

Run a Replicated Stateful Application

Scale a StatefulSet

Delete a StatefulSet

Force Delete StatefulSet Pods

Horizontal Pod Autoscaling

HorizontalPodAutoscaler Walkthrough

Specifying a Disruption Budget for your Application

Accessing the Kubernetes API from a Pod

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.9. To check the version, enter `kubectl version`.

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:1.14.2 Docker image:

application/deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the
template
  template:
    metadata:
      labels:
```

```
      app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

1. Create a Deployment based on the YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/
   deployment.yaml
   ```

2. Display information about the Deployment:

   ```
   kubectl describe deployment nginx-deployment
   ```

   The output is similar to this:

   ```
   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:1.14.2
       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
   ```

The output is similar to this:

```
 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 1.16.1.

application/deployment-update.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1 # Update the version of nginx from 
1.14.2 to 1.16.1
        ports:
        - containerPort: 80
```

1. Apply the new YAML file:

   ```
     kubectl apply -f https://k8s.io/examples/application/
   deployment-update.yaml
   ```

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:

application/deployment-scale.yaml

```
apiVersion: apps/v1
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:1.14.2
        ports:
        - containerPort: 80
```

1. Apply the new YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/
   deployment-scale.yaml
   ```

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 using a ReplicationController.

What's next

- Learn more about Deployment objects.

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

- 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.

application/mysql/mysql-deployment.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
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
```

application/mysql/mysql-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

1. Deploy the PV and PVC of the YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/mysql/mysql-pv.yaml
   ```

2. Deploy the contents of the YAML file:

```
 kubectl apply -f https://k8s.io/examples/application/mysql/
mysql-deployment.yaml
```

3. Display information about the Deployment:

```
 kubectl describe deployment mysql
```

The output is similar to this:

```
 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 unavailable
 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 in the same namespace)
     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        Reason           Message
   ---------    --------    -----    ----
-------------    --------    ------           -------
   33s          33s         1        {deployment-
controller }             Normal      ScalingReplicaSet
Scaled up replica set mysql-63082529 to 1
```

4. List the pods created by the Deployment:

```
kubectl get pods -l app=mysql
```

The output is similar to this:

```
NAME                   READY     STATUS    RESTARTS   AGE
mysql-63082529-2z3ki   1/1       Running   0          3m
```

5. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim
```

The output is similar to this:

```
Name:         mysql-pv-claim
Namespace:    default
StorageClass:
Status:       Bound
Volume:       mysql-pv-volume
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 mysql -ppassword
```

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 Pending, pod ready: false
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
kubectl delete pv mysql-pv-volume
```

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. This application is a replicated MySQL database. The example topology has a single primary server and multiple replicas, using asynchronous row-based replication.

Note: This is not a production configuration. MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

- 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.
- You are using the default namespace or another namespace that does not contain any conflicting objects.

Objectives

- Deploy a replicated MySQL topology with a StatefulSet controller.
- Send MySQL client traffic.
- Observe resistance to downtime.
- Scale the StatefulSet up and down.

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:

application/mysql/mysql-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
```

```yaml
metadata:
  name: mysql
  labels:
    app: mysql
data:
  primary.cnf: |
    # Apply this config only on the primary.
    [mysqld]
    log-bin
    datadir=/var/lib/mysql/mysql
  replica.cnf: |
    # Apply this config only on replicas.
    [mysqld]
    super-read-only
    datadir=/var/lib/mysql/mysql
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-configmap.yaml
```

This ConfigMap provides `my.cnf` overrides that let you independently control configuration on the primary MySQL server and replicas. In this case, you want the primary server to be able to serve replication logs to replicas and you want replicas 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:

application/mysql/mysql-services.yaml

```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
```

```yaml
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the primary:
mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-
services.yaml
```

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 primary MySQL server and all replicas.

Note that only read queries can use the load-balanced Client Service. Because there is only one primary MySQL server, clients should connect directly to the primary MySQL Pod (through its DNS entry within the Headless Service) to execute writes.

StatefulSet

Finally, create the StatefulSet from the following YAML configuration file:

application/mysql/mysql-statefulset.yaml

```yaml
apiVersion: apps/v1
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 [[ $ordinal -eq 0 ]]; then
            cp /mnt/config-map/primary.cnf /mnt/conf.d/
          else
            cp /mnt/config-map/replica.cnf /mnt/conf.d/
          fi
        volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
      - name: clone-mysql
        image: gcr.io/google-samples/xtrabackup:1.0
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on primary (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || 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/lib/mysql
          # Prepare the backup.
          xtrabackup --prepare --target-dir=/var/lib/mysql
```

```yaml
        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 && "x$
(<xtrabackup_slave_info)" != "x" ]]; then
            # XtraBackup already generated a partial "CHANGE
MASTER TO" query
            # because we're cloning from an existing replica.
(Need to remove the tailing semicolon!)
            cat xtrabackup_slave_info | sed -E 's/;$//g' >
change_master_to.sql.in
            # Ignore xtrabackup_binlog_info in this case (it's
useless).
            rm -f xtrabackup_slave_info xtrabackup_binlog_info
          elif [[ -f xtrabackup_binlog_info ]]; then
            # We're cloning directly from primary. Parse binlog
position.
            [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+
(.*?)$ ]] || exit 1
            rm -f xtrabackup_binlog_info xtrabackup_slave_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"
            mysql -h 127.0.0.1 \
                  -e "$(<change_master_to.sql.in), \
                          MASTER_HOST='mysql-0.mysql', \
                          MASTER_USER='root', \
                          MASTER_PASSWORD='', \
                          MASTER_CONNECT_RETRY=10; \
                        START SLAVE;" || exit 1
            # In case of container restart, attempt this at-most-
once.
            mv change_master_to.sql.in change_master_to.sql.orig
          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 --user=root"
```

```
        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
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-
statefulset.yaml
```

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>`, which 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 `primary.cnf` or `replica.cnf` from the ConfigMap by copying the contents into `conf.d`. Because the example topology consists of a single primary MySQL server and any number of replicas, the script assigns ordinal `0` to be the primary server, and everyone else to be replicas. Combined with the StatefulSet controller's deployment order guarantee, this ensures the primary MySQL server is Ready before creating replicas, so they can begin replicating.

Cloning existing data

In general, when a new Pod joins the set as a replica, it must assume the primary MySQL server 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 replica 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 primary server.

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 primary MySQL server, 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 replica. 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 replica begins replication, it remembers its primary MySQL server and reconnects automatically if the server restarts or the connection dies. Also, because replicas look for the primary server at its stable DNS name (`mysql-0.mysql`), they automatically find the primary server 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 primary MySQL server (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: false
+---------+
| 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()'; done"
```

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 replicas 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-node-9l2t
```

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-emptydir-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-node-9l2t
[...]
mysql-2   0/2     Pending         0          0s
<none>          kubernetes-node-fjlm
mysql-2   0/2     Init:0/2        0          0s
<none>          kubernetes-node-fjlm
mysql-2   0/2     Init:1/2        0          20s
10.244.5.32   kubernetes-node-fjlm
mysql-2   0/2     PodInitializing 0          21s
10.244.5.32   kubernetes-node-fjlm
mysql-2   1/2     Running         0          22s
10.244.5.32   kubernetes-node-fjlm
mysql-2   2/2     Running         0          30s
10.244.5.32   kubernetes-node-fjlm
```

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 replicas

With MySQL replication, you can scale your read query capacity by adding replicas. 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=N
ever --\
  mysql -h mysql-3.mysql -e "SELECT * FROM test.messages"
```

```
Waiting for pod default/mysql-client to be running, status is
Pending, pod ready: false
+---------+
| 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
ACCESSMODES   AGE
data-mysql-0   Bound     pvc-8acbf5dc-
b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-1   Bound     pvc-8ad39820-
b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-2   Bound     pvc-8ad69a6d-
b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-3   Bound     pvc-50043c45-
b1c5-11e6-93fa-42010a800002   10Gi       RWO           2m
data-mysql-4   Bound     pvc-500a9957-
b1c5-11e6-93fa-42010a800002   10Gi       RWO           2m
```

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

- Learn more about scaling a StatefulSet.
- Learn more about debugging a StatefulSet.
- Learn more about deleting a StatefulSet.
- Learn more about force deleting StatefulSet Pods.
- Look in the Helm Charts repository for other stateful application examples.

Scale a StatefulSet

This task shows how to scale a StatefulSet. Scaling a StatefulSet refers to increasing or decreasing the number of replicas.

Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later. To check your version of Kubernetes, run `kubectl version`.

- Not all stateful applications scale nicely. If you are unsure about whether to scale your StatefulSets, see StatefulSet concepts or StatefulSet tutorial for further information.
- You should perform scaling only when you are confident that your stateful application cluster is completely healthy.

Scaling StatefulSets

Use kubectl to scale StatefulSets

First, find the StatefulSet you want to scale.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replicas>
```

Make in-place updates on your StatefulSets

Alternatively, you can do in-place updates on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply`, 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-replicas>}}'
```

Troubleshooting

Scaling down does not 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.

If spec.replicas > 1, Kubernetes cannot determine the reason for an unhealthy Pod. It might be the result of a permanent fault or of a transient fault. A transient fault can be caused by a restart required by upgrading or maintenance.

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 or 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 are sure that your stateful application cluster is completely healthy.

What's next

- Learn more about deleting a StatefulSet.

Delete a StatefulSet

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>
```

When deleting a StatefulSet through `kubectl`, the StatefulSet scales down to 0. All Pods that are part of this workload are also deleted. If you want to delete only the StatefulSet and not the Pods, use `--cascade=orphan`. For example:

```
kubectl delete -f <file.yaml> --cascade=orphan
```

By passing `--cascade=orphan` 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 terminated 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 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.t
erminationGracePeriodSeconds}}')
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 Force Delete 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.

A Pod is not deleted automatically when 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
```

If even after these commands the pod is stuck on `Unknown` state, use the following command to remove the pod from the cluster:

```
kubectl patch pod <pod> -p '{"metadata":{"finalizers":null}}'
```

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.

Horizontal Pod Autoscaling

In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

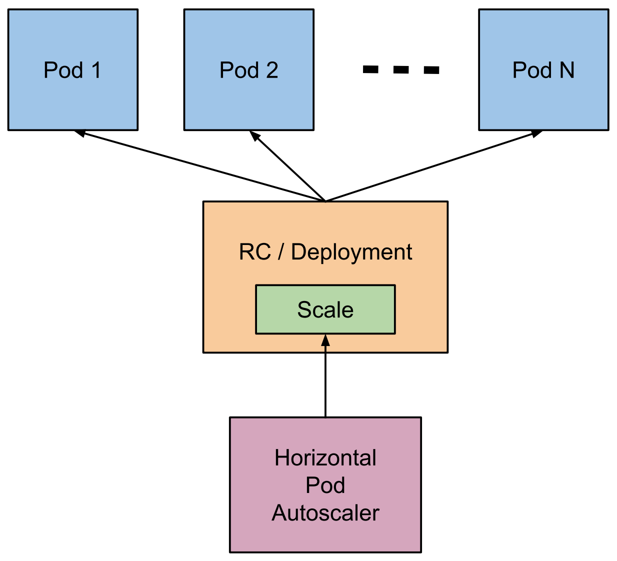
If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a DaemonSet.)

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.

There is walkthrough example of using horizontal pod autoscaling.

How does a HorizontalPodAutoscaler work?



HorizontalPodAutoscaler controls the scale of a Deployment and its ReplicaSet

Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently (it is not a continuous process). The interval is set by the `--horizontal-pod-autoscaler-sync-period` parameter to the `kube-controller-manager` (and the default interval is 15 seconds).

Once during each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition. The controller manager finds the target resource defined by the `scaleTargetRef`, then selects the pods based on the target resource's `.spec.selector` labels, and 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 algorithm details section below for more information 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 and external metrics, a single metric is fetched, which describes the object in question. This metric is compared to the target value, to produce a ratio as above. In the `autoscaling/v2` API version, this value can optionally be divided by the number of Pods before the comparison is made.

The common use for HorizontalPodAutoscaler is to configure it to fetch metrics from aggregated APIs (`metrics.k8s.io`, `custom.metrics.k8s.io`, or `external.metrics.k8s.io`). The `metrics.k8s.io` API is usually provided by an add-on named Metrics Server, which needs to be launched separately. For more information about resource metrics, see Metrics Server.

Support for metrics APIs explains the stability guarantees and support status for these different APIs.

The HorizontalPodAutoscaler controller accesses corresponding workload resources that support scaling (such as Deployments and StatefulSet). These resources each have a subresource named `scale`, an interface that allows you to dynamically set the number of replicas and examine each of their current states. For general information about subresources in the Kubernetes API, see Kubernetes API Concepts.

Algorithm details

From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue /
desiredMetricValue )]
```

For example, if the current metric value is `200m`, and the desired value is `100m`, the number of replicas will be doubled, since `200.0 / 100.0 == 2.0` If the current value is instead `50m`, you'll halve the number of replicas, since `50.0 / 100.0 == 0.5`. The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

When a `targetAverageValue` or `targetAverageUtilization` is specified, the `currentMetricValue` is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target.

Before checking the tolerance and deciding on the final values, the control plane also considers whether any metrics are missing, and how many Pods are Ready. All Pods with a deletion timestamp set (objects with a deletion timestamp are in the process of being shut down / removed) are ignored, and all failed Pods are discarded.

If a particular Pod is missing metrics, it is set aside for later; Pods with missing metrics will be used to adjust the final scaling amount.

When scaling on CPU, if any pod has yet to become ready (it's still initializing, or possibly is unhealthy) *or* the most recent metric point for the pod was before it became ready, that pod is set aside as well.

Due to technical constraints, the HorizontalPodAutoscaler controller cannot exactly determine the first time a pod becomes ready when determining whether to set aside certain CPU metrics. Instead, it considers a Pod "not yet ready" if it's unready and transitioned to unready within a short, configurable window of time since it started. This value is configured with the `--horizontal-pod-autoscaler-initial-readiness-delay` flag, and its default is 30 seconds. Once a pod has become ready, it considers any transition to ready to be the first if it occurred within a longer, configurable time since it started. This value is configured with the `--horizontal-pod-autoscaler-cpu-initialization-period` flag, and its default is 5 minutes.

The `currentMetricValue / desiredMetricValue` base scale ratio is then calculated using the remaining pods not set aside or discarded from above.

If there were any missing metrics, the control plane recomputes the average more conservatively, assuming those pods were consuming 100% of the desired value in case of a scale down, and 0% in case of a scale up. This dampens the magnitude of any potential scale.

Furthermore, if any not-yet-ready pods were present, and the workload would have scaled up without factoring in missing metrics or not-yet-ready pods, the controller conservatively assumes that the not-yet-ready pods are consuming 0% of the desired metric, further dampening the magnitude of a scale up.

After factoring in the not-yet-ready pods and missing metrics, the controller recalculates the usage ratio. If the new ratio reverses the scale direction, or is within the tolerance, the controller doesn't take any scaling action. In other cases, the new ratio is used to decide any change to the number of Pods.

Note that the *original* value for the average utilization is reported back via the HorizontalPodAutoscaler status, without factoring in the not-yet-ready pods or missing metrics, even when the new usage ratio is used.

If multiple metrics are specified in a HorizontalPodAutoscaler, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. If any of these metrics cannot be converted into a desired replica count (e.g. due to an error fetching the metrics from the metrics APIs) and a scale down is suggested by the metrics which can be fetched, scaling is skipped. This means that the HPA is still capable of scaling up if one or more metrics give a `desiredReplicas` greater than the current value.

Finally, right before HPA scales the target, the scale recommendation is recorded. The controller considers all recommendations within a configurable window choosing the highest recommendation from within that window. This value can be configured using the `--horizontal-pod-autoscaler-downscale-stabilization` flag, which defaults to 5 minutes. This means that scaledowns will occur gradually, smoothing out the impact of rapidly fluctuating metric values.

API Object

The Horizontal Pod Autoscaler is an API resource in the Kubernetes `autoscaling` API group. The current stable version can be found in the `autoscaling/v2` API version which includes support for scaling on memory and custom metrics. The new fields introduced in `autoscaling/v2` are preserved as annotations when working with `autoscaling/v1`.

When you create a HorizontalPodAutoscaler API object, make sure the name specified is a valid DNS subdomain name. More details about the API object can be found at HorizontalPodAutoscaler Object.

Stability of workload scale

When managing the scale of a group of replicas using the HorizontalPodAutoscaler, it is possible that the number of replicas keeps fluctuating frequently due to the dynamic nature of the metrics evaluated. This is sometimes referred to as *thrashing*, or *flapping*. It's similar to the concept of *hysteresis* in cybernetics.

Autoscaling during rolling update

Kubernetes lets you perform a rolling update on a Deployment. In that case, the Deployment manages the underlying ReplicaSets for you. When you configure autoscaling for a Deployment, you bind a HorizontalPodAutoscaler to a single Deployment. The HorizontalPodAutoscaler manages the `replicas` field of the Deployment. The deployment controller is responsible for setting the `replicas` of the underlying ReplicaSets so that they add up to a suitable number during the rollout and also afterwards.

If you perform a rolling update of a StatefulSet that has an autoscaled number of replicas, the StatefulSet directly manages its set of Pods (there is no intermediate resource similar to ReplicaSet).

Support for resource metrics

Any HPA target can be scaled based on the resource usage of the pods in the scaling target. When defining the pod specification the resource requests like `cpu` and `memory` should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```yaml
type: Resource
resource:
  name: cpu
  target:
    type: Utilization
    averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%. Utilization is the ratio between the current usage of resource to the requested resources of the pod. See Algorithm for more details about how the utilization is calculated and averaged.

Note: Since the resource usages of all the containers are summed up the total pod utilization may not accurately represent the individual container resource usage. This could lead to situations where a single container might be running with high usage and the HPA will not scale out because the overall pod usage is still within acceptable limits.

Container resource metrics

FEATURE STATE: `Kubernetes v1.20 [alpha]`

The HorizontalPodAutoscaler API also supports a container metric source where the HPA can track the resource usage of individual containers across a set of Pods, in order to scale the target resource. This lets you configure scaling thresholds for the containers that matter most in a particular Pod. For example, if you have a web application and a logging sidecar, you can scale based on the resource use of the web application, ignoring the sidecar container and its resource use.

If you revise the target resource to have a new Pod specification with a different set of containers, you should revise the HPA spec if that newly added container should also be used for scaling. If the specified container in the metric source is not present or only present in a subset of the pods then those pods are ignored and the recommendation is recalculated. See Algorithm for more details about the calculation. To use container resources for autoscaling define a metric source as follows:

```yaml
type: ContainerResource
containerResource:
  name: cpu
  container: application
  target:
    type: Utilization
    averageUtilization: 60
```

In the above example the HPA controller scales the target such that the average utilization of the cpu in the `application` container of all the pods is 60%.

Note:

If you change the name of a container that a HorizontalPodAutoscaler is tracking, you can make that change in a specific order to ensure scaling remains available and effective whilst the change is being applied. Before you update the resource that defines the container (such as a Deployment), you should update the associated HPA to track both the new and old container names. This way, the HPA is able to calculate a scaling recommendation throughout the update process.

Once you have rolled out the container name change to the workload resource, tidy up by removing the old container name from the HPA specification.

Scaling on custom metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can configure a HorizontalPodAutoscaler to scale based on a custom metric (that is not built in to Kubernetes or any Kubernetes component). The HorizontalPodAutoscaler controller then queries for these custom metrics from the Kubernetes API.

See Support for metrics APIs for the requirements.

Scaling on multiple metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can specify multiple metrics for a HorizontalPodAutoscaler to scale on. Then, the HorizontalPodAutoscaler controller evaluates each metric, and proposes a new scale based on that metric. The HorizontalPodAutoscaler takes the maximum scale recommended for each metric and sets the workload to that size (provided that this isn't larger than the overall maximum that you configured).

Support for metrics APIs

By default, the HorizontalPodAutoscaler controller retrieves metrics from a series of APIs. In order for it to access these APIs, cluster administrators must ensure that:

- The API aggregation layer is enabled.
- The corresponding APIs are registered:
  - For resource metrics, this is the `metrics.k8s.io` API, generally provided by metrics-server. It can be launched as a cluster add-on.
  - For custom metrics, this is the `custom.metrics.k8s.io` API. It's provided by "adapter" API servers provided by metrics solution vendors. Check with your metrics pipeline to see if there is a Kubernetes metrics adapter available.
  - For external metrics, this is the `external.metrics.k8s.io` API. It may be provided by the custom metrics adapters provided above.

For more information on these different metrics paths and how they differ please see the relevant design proposals for the HPA V2, custom.metrics.k8s.io and external.metrics.k8s.io.

For examples of how to use them see the walkthrough for using custom metrics and the walkthrough for using external metrics.

Configurable scaling behavior

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

If you use the `v2` HorizontalPodAutoscaler API, you can use the `behavior` field (see the API reference) to configure separate scale-up and scale-down behaviors. You specify these behaviours by setting `scaleUp` and / or `scaleDown` under the `behavior` field.

You can specify a *stabilization window* that prevents flapping the replica count for a scaling target. Scaling policies also let you controls the rate of change of replicas while scaling.

Scaling policies

One or more scaling policies can be specified in the `behavior` section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default. The following example shows this behavior while scaling down:

```yaml
behavior:
  scaleDown:
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
```

`periodSeconds` indicates the length of time in the past for which the policy must hold true. The first policy *(Pods)* allows at most 4 replicas to be scaled down in one minute. The second policy *(Percent)* allows at most 10% of the current replicas to be scaled down in one minute.

Since by default the policy which allows the highest amount of change is selected, the second policy will only be used when the number of pod replicas is more than 40. With 40 or less replicas, the first policy will be applied. For instance if there are 80 replicas and the target has to be scaled down to 10 replicas then during the first step 8 replicas will be reduced. In the next iteration when the number of replicas is 72, 10% of the pods is 7.2 but the number is rounded up to 8. On each loop of the autoscaler controller the number of pods to be change is re-calculated based on the number of current replicas. When the number of replicas falls below 40 the first policy *(Pods)* is applied and 4 replicas will be reduced at a time.

The policy selection can be changed by specifying the `selectPolicy` field for a scaling direction. By setting the value to `Min` which would select the policy which allows the smallest change in the replica count. Setting the value to `Disabled` completely disables scaling in that direction.

Stabilization window

The stabilization window is used to restrict the flapping of replicas count when the metrics used for scaling keep fluctuating. The autoscaling algorithm uses this window to infer a previous desired state and avoid unwanted changes to workload scale.

For example, in the following example snippet, a stabilization window is specified for `scaleDown`.

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

When the metrics indicate that the target should be scaled down the algorithm looks into previously computed desired states, and uses the highest value from the specified interval. In the above example, all desired states from the past 5 minutes will be considered.

This approximates a rolling maximum, and avoids having the scaling algorithm frequently remove Pods only to trigger recreating an equivalent Pod just moments later.

Default Behavior

To use the custom scaling not all fields have to be specified. Only values which need to be customized can be specified. These custom values are merged with default values. The default values match the existing behavior in the HPA algorithm.

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

For scaling down the stabilization window is *300* seconds (or the value of the `--horizontal-pod-autoscaler-downscale-stabilization` flag if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas will be added every 15 seconds till the HPA reaches its steady state.

Example: change downscale stabilization window

To provide a custom downscale stabilization window of 1 minute, the following behavior would be added to the HPA:

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 60
```

Example: limit scale down rate

To limit the rate at which pods are removed by the HPA to 10% per minute, the following behavior would be added to the HPA:

```yaml
behavior:
  scaleDown:
    policies:
```

```
    - type: Percent
      value: 10
      periodSeconds: 60
```

To ensure that no more than 5 Pods are removed per minute, you can add a second scale-down policy with a fixed size of 5, and set `selectPolicy` to minimum. Setting `selectPolicy` to `Min` means that the autoscaler chooses the policy that affects the smallest number of Pods:

```
behavior:
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
    - type: Pods
      value: 5
      periodSeconds: 60
    selectPolicy: Min
```

Example: disable scale down

The `selectPolicy` value of `Disabled` turns off scaling the given direction. So to prevent downscaling the following policy would be used:

```
behavior:
  scaleDown:
    selectPolicy: Disabled
```

Support for HorizontalPodAutoscaler in kubectl

HorizontalPodAutoscaler, like every API resource, is supported in a standard way by `kubectl`. You can create a new autoscaler using `kubectl create` command. You can list autoscalers by `kubectl get hpa` or get detailed description by `kubectl describe hpa`. Finally, you can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for creating a HorizontalPodAutoscaler object. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for ReplicaSet *foo*, with target CPU utilization set to `80%` and the number of replicas between 2 and 5.

Implicit maintenance-mode deactivation

You can implicitly deactivate the HPA for a target without the need to change the HPA configuration itself. If the target's desired replica count is set to 0, and the HPA's minimum replica count is greater than 0, the HPA stops adjusting the target (and sets the `ScalingActive` Condition on itself to `false`) until you reactivate it by manually adjusting the target's desired replica count or HPA's minimum replica count.

Migrating Deployments and StatefulSets to horizontal autoscaling

When an HPA is enabled, it is recommended that the value of `spec.replicas` of the Deployment and / or StatefulSet be removed from their manifest(s). If this isn't done, any time a change to that object is applied, for example via `kubectl apply -f deployment.yaml`, this will instruct Kubernetes to scale the current number of Pods to the value of the `spec.replicas` key. This may not be desired and could be troublesome when an HPA is active.

Keep in mind that the removal of `spec.replicas` may incur a one-time degradation of Pod counts as the default value of this key is 1 (reference Deployment Replicas). Upon the update, all Pods except 1 will begin their termination procedures. Any deployment application afterwards will behave as normal and respect a rolling update configuration as desired. You can avoid this degradation by choosing one of the following two methods based on how you are modifying your deployments:

- Client Side Apply (this is the default)
- Server Side Apply

1. `kubectl apply edit-last-applied deployment/<deployment_name>`
2. In the editor, remove `spec.replicas`. When you save and exit the editor, `kubectl` applies the update. No changes to Pod counts happen at this step.
3. You can now remove `spec.replicas` from the manifest. If you use source code management, also commit your changes or take whatever other steps for revising the source code are appropriate for how you track updates.
4. From here on out you can run `kubectl apply -f deployment.yaml`

When using the Server-Side Apply you can follow the transferring ownership guidelines, which cover this exact use case.

What's next

If you configure autoscaling in your cluster, you may also want to consider running a cluster-level autoscaler such as Cluster Autoscaler.

For more information on HorizontalPodAutoscaler:

- Read a walkthrough example for horizontal pod autoscaling.
- Read documentation for `kubectl autoscale`.
- If you would like to write your own custom metrics adapter, check out the boilerplate to get started.
- Read the API reference for HorizontalPodAutoscaler.

HorizontalPodAutoscaler Walkthrough

A HorizontalPodAutoscaler (HPA for short) automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

This document walks you through an example of enabling HorizontalPodAutoscaler to automatically manage scale for an example web app. This example workload is Apache httpd running some PHP code.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version 1.23. To check the version, enter `kubectl version`. If you're running an older release of Kubernetes, refer to the version of the documentation for that release (see available documentation versions).

To follow this walkthrough, you also need to use a cluster that has a Metrics Server deployed and configured. The Kubernetes Metrics Server collects resource metrics from the kubelets in your cluster, and exposes those metrics through the Kubernetes API, using an APIService to add new kinds of resource that represent metric readings.

To learn how to deploy the Metrics Server, see the metrics-server documentation.

Run and expose php-apache server

To demonstrate a HorizontalPodAutoscaler, you will first make a custom container image that uses the `php-apache` image from Docker Hub as its starting point. The Dockerfile is ready-made for you, and has the following content:

```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

This code defines a simple index.php page that performs some CPU intensive computations, in order to simulate load in your cluster.

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

Once you have made that container image, start a Deployment that runs a container using the image you made, and expose it as a Service using the following manifest:

application/php-apache.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

To do so, run the following command:

```
kubectl apply -f https://k8s.io/examples/application/php-
apache.yaml
```

```
deployment.apps/php-apache created
service/php-apache created
```

Create the HorizontalPodAutoscaler

Now that the server is running, create the autoscaler using `kubectl`. There is [`kubectl autoscale`]() subcommand, part of `kubectl`, that helps you do this.

You will shortly run a command that creates a HorizontalPodAutoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache Deployment that you created in the first step of these instructions.

Roughly speaking, the HPA controller will increase and decrease the number of replicas (by updating the Deployment) to maintain an average CPU utilization across all Pods of 50%. The Deployment then updates the ReplicaSet - this is part of how all Deployments work in Kubernetes - and then the ReplicaSet either adds or removes Pods based on the change to its `.spec`.

Since each pod requests 200 milli-cores by `kubectl run`, this means an average CPU usage of 100 milli-cores. See Algorithm details for more details on the algorithm.

Create the HorizontalPodAutoscaler:

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1
--max=10
```

```
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

You can check the current status of the newly-made HorizontalPodAutoscaler, by running:

```
# You can use "hpa" or "horizontalpodautoscaler"; either name
works OK.
kubectl get hpa
```

The output is similar to:

```
NAME         REFERENCE                     TARGET    MINPODS
MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   0% / 50%  1
10        1          18s
```

(if you see other HorizontalPodAutoscalers with different names, that means they already existed, and isn't usually a problem).

Please note that the current CPU consumption is 0% as there are no clients sending requests to the server (the `TARGET` column shows the average across all the Pods controlled by the corresponding deployment).

Increase the load

Next, see how the autoscaler reacts to increased load. To do this, you'll start a different Pod to act as a client. The container within the client Pod runs in an infinite loop, sending queries to the php-apache service.

```
# Run this in a separate terminal
# so that the load generation continues and you can carry on
with the rest of the steps
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --
restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O-
http://php-apache; done"
```

Now run:

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

Within a minute or so, you should see the higher CPU load; for example:

```
NAME         REFERENCE                     TARGET      MINPODS
MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   305% / 50%  1
10        1          3m
```

and then, more replicas. For example:

```
NAME         REFERENCE                     TARGET      MINPODS
MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   305% / 50%  1
10        7          3m
```

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
```

You should see the replica count matching the figure from the HorizontalPodAutoscaler

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   7/7      7            7           19m
```

Note: 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.

Stop generating load

To finish the example, stop sending the load.

In the terminal where you created the Pod that runs a `busybox` image, terminate the load generation by typing `<Ctrl> + C`.

Then verify the result state (after a minute or so):

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

The output is similar to:

```
NAME         REFERENCE                     TARGET
MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   0% / 50%
1         10        1          11m
```

and the Deployment also shows that it has scaled down:

```
kubectl get deployment php-apache
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           27m
```

Once CPU utilization dropped to 0, the HPA automatically scaled the number of replicas back down to 1.

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/v2` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2` form:

```
kubectl get hpa php-apache -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:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      current:
        averageUtilization: 0
        averageValue: 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 the `metrics.k8s.io` API is available.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value, by using a `target.type` of `AverageValue` instead of `Utilization`, and setting the corresponding `target.averageValue` field instead of the `target.averageUtilization`.

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* support a `target` type of `AverageValue`.

Pod metrics are specified using a metric block like this:

```yaml
type: Pods
pods:
  metric:
    name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing Pods. The metrics are not necessarily fetched from the object; they only describe it. Object metrics support `target` types of both `Value` and `AverageValue`. With `Value`, the target is compared directly to the returned metric from the API. With `AverageValue`, the value returned from the custom metrics API is divided by the number of Pods before being compared to the target. The following example is the YAML representation of the `requests-per-second` metric.

```yaml
type: Object
object:
  metric:
    name: requests-per-second
  describedObject:
    apiVersion: networking.k8s.io/v1
    kind: Ingress
    name: main-route
  target:
    type: Value
    value: 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:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Pods
    pods:
      metric:
        name: packets-per-second
      target:
        type: AverageValue
        averageValue: 1k
  - type: Object
    object:
      metric:
        name: requests-per-second
      describedObject:
        apiVersion: networking.k8s.io/v1
        kind: Ingress
        name: main-route
      target:
        type: Value
        value: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
    current:
      averageUtilization: 0
      averageValue: 0
  - type: Object
    object:
      metric:
        name: requests-per-second
      describedObject:
        apiVersion: networking.k8s.io/v1
        kind: Ingress
        name: main-route
      current:
        value: 10k
```

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.

Autoscaling on more specific metrics

Many metrics pipelines allow you to describe metrics either by name or by a set of additional descriptors called *labels*. For all non-resource metric types (pod, object, and external, described below), you can specify an additional label selector which is passed to your metric pipeline. For instance, if you collect a metric `http_requests` with the `verb` label, you can specify the following metric block to scale only on GET requests:

```
type: Object
object:
  metric:
    name: http_requests
    selector: {matchLabels: {verb: GET}}
```

This selector uses the same syntax as the full Kubernetes label selectors. The monitoring pipeline determines how to collapse multiple series into a single value, if the name and selector match multiple series. The selector is additive, and cannot select metrics that describe objects that are **not** the target object (the target pods in the case of the `Pods` type, and the described object in the case of the `Object` type).

Autoscaling on metrics not related to Kubernetes objects

Applications running on Kubernetes may need to autoscale based on metrics that don't have an obvious relationship to any object in the Kubernetes cluster, such as metrics describing a hosted service with no direct correlation to Kubernetes namespaces. In Kubernetes 1.10 and later, you can address this use case with *external metrics*.

Using external metrics requires knowledge of your monitoring system; the setup is similar to that required when using custom metrics. External metrics allow you to autoscale your cluster based on any metric available in your monitoring system. Provide a `metric` block with a `name` and `selector`, as above, and use the `External` metric type instead of `Object`. If multiple time series are matched by the `metricSelector`, the sum of their values is used by the HorizontalPodAutoscaler. External metrics support both the `Value` and `AverageValue` target types, which function exactly the same as when you use the `Object` type.

For example if your application processes tasks from a hosted queue service, you could add the following section to your HorizontalPodAutoscaler manifest to specify that you need one worker per 30 outstanding tasks.

```
- type: External
  external:
    metric:
      name: queue_messages_ready
      selector:
        matchLabels:
          queue: "worker_tasks"
    target:
      type: AverageValue
      averageValue: 30
```

When possible, it's preferable to use the custom metric target types instead of external metrics, since it's easier for cluster administrators to secure the custom metrics API. The external metrics API potentially allows access to any metric, so cluster administrators should take care when exposing it.

Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2` 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 sufficiently old as to warrant a new scale
  ScalingActive         True    ValidMetricFound        the HPA
was able to successfully calculate a replica count from pods
metric http_requests
  ScalingLimited        False   DesiredWithinRange      the
desired replica count is within the acceptable range
Events:
```

For this HorizontalPodAutoscaler, you 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, `ScalingLimited`, 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.

Quantities

All metrics in the HorizontalPodAutoscaler and metrics APIs are specified using a special whole-number notation known in Kubernetes as a quantity. For example, the quantity `10500m` would be written as `10.5` in decimal notation. The metrics APIs will return whole numbers without a suffix when possible, and will generally return quantities in milli-units otherwise. This means you might see your metric value fluctuate between `1` and `1500m`, or `1` and `1.5` when written in decimal notation.

Other possible scenarios

Creating the autoscaler declaratively

Instead of using `kubectl autoscale` command to create a HorizontalPodAutoscaler imperatively we can use the following manifest to create it declaratively:

application/hpa/php-apache.yaml

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Then, create the autoscaler by executing the following command:

```
kubectl create -f https://k8s.io/examples/application/hpa/php-apache.yaml
```

```
horizontalpodautoscaler.autoscaling/php-apache created
```

Specifying a Disruption Budget for your Application

FEATURE STATE: `Kubernetes v1.21 [stable]`

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

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

- 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`.

From version 1.15 PDBs support custom controllers where the scale subresource is enabled.

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.

Rounding logic when specifying percentages

Values for `minAvailable` or `maxUnavailable` can be expressed as integers or as a percentage.

- When you specify an integer, it represents a number of Pods. For instance, if you set `minAvailable` to 10, then 10 Pods must always be available, even during a disruption.
- When you specify a percentage by setting the value to a string representation of a percentage (eg. `"50%"`), it represents a percentage of total Pods. For instance, if you set `maxUnavailable` to `"50%"`, then only 50% of the Pods can be unavailable during a disruption.

When you specify the value as a percentage, it may not map to an exact number of Pods. For example, if you have 7 Pods and you set `minAvailable` to `"50%"`, it's not immediately obvious whether that means 3 Pods or 4 Pods must be available. Kubernetes rounds up to the nearest integer, so in this case, 4 Pods must be available. You can examine the code that controls this behavior.

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.

Note: The behavior for an empty selector differs between the policy/v1beta1 and policy/v1 APIs for PodDisruptionBudgets. For policy/v1beta1 an empty selector matches zero pods, while for policy/v1 an empty selector matches every pod in the namespace.

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 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 controllerâ€”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.

If you set `maxUnavailable` to 0% or 0, or you set `minAvailable` to 100% or the number of replicas, you are requiring zero voluntary evictions. When you set zero voluntary evictions for a workload object such as ReplicaSet, then you cannot successfully drain a Node running one of those Pods. If you try to drain a Node where an unevictable Pod is running, the drain never completes. 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:

policy/zookeeper-pod-disruption-budget-minavailable.yaml

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

Example PDB Using maxUnavailable:

policy/zookeeper-pod-disruption-budget-maxunavailable.yaml

```yaml
apiVersion: policy/v1
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 or update the PDB object using kubectl.

```
kubectl apply -f mypdb.yaml
```

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   MAX UNAVAILABLE   ALLOWED DISRUPTIONS
AGE
zk-pdb   2               N/A               0
7s
```

If there are matching pods (say, 3), then you would see something like this:

```
kubectl get poddisruptionbudgets
```

```
NAME     MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS
AGE
zk-pdb   2               N/A               1
7s
```

The non-zero value for `ALLOWED DISRUPTIONS` means that the disruption controller has seen the pods, counted the matching pods, and updated 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/v1
kind: PodDisruptionBudget
metadata:
  annotations:
â€¦
  creationTimestamp: "2020-03-04T04:22:56Z"
  generation: 1
  name: zk-pdb
â€¦
status:
  currentHealthy: 3
  desiredHealthy: 2
  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. The eviction API will disallow eviction of any pod covered by multiple PDBs, so most users will want to avoid overlapping selectors. One reasonable use of overlapping PDBs is when pods are being transitioned from one PDB to another.

Accessing the Kubernetes API from a Pod

This guide demonstrates how to access the Kubernetes API from within a 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 API from within a Pod

When accessing the API from within a Pod, locating and authenticating to the API server are slightly different to the external client case.

The easiest way to use the Kubernetes API from a Pod is to use one of the official client libraries. These libraries can automatically discover the API server and authenticate.

Using Official Client Libraries

From within a Pod, the recommended ways to connect to the Kubernetes API are:

- For a Go client, use the official Go client library. The `rest.InClusterConfig()` function handles API host discovery and authentication automatically. See an example here.
- For a Python client, use the official Python client library. The `config.load_incluster_config()` function handles API host discovery and authentication automatically. See an example here.
- There are a number of other libraries available, please refer to the Client Libraries page.

In each case, the service account credentials of the Pod are used to communicate securely with the API server.

Directly accessing the REST API

While running in a Pod, the Kubernetes apiserver is accessible via a Service named `kubernetes` in the `default` namespace. Therefore, Pods can use the `kubernetes.default.svc` hostname to query the API server. Official client libraries do this automatically.

The recommended way to authenticate to the API server is with a service account credential. By default, 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 API server.

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.

Using kubectl proxy

If you would like to query the API without an official client library, you can run `kubectl proxy` as the command of a new sidecar container in the Pod. This way, `kubectl proxy` will authenticate to the API and expose it on the `localhost` interface of the Pod, so that other containers in the Pod can use it directly.

Without using a proxy

It is possible to avoid using the kubectl proxy by passing the authentication token directly to the API server. The internal certificate secures the connection.

```
# Point to the internal API server hostname
APISERVER=https://kubernetes.default.svc

# Path to ServiceAccount token
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# Read this Pod's namespace
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# Read the ServiceAccount bearer token
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# Reference the internal certificate authority (CA)
CACERT=${SERVICEACCOUNT}/ca.crt

# Explore the API with TOKEN
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}"
-X GET ${APISERVER}/api
```

The output will be similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

Run Jobs

Run Jobs using parallel processing.

Running Automated Tasks with a CronJob

Coarse Parallel Processing Using a Work Queue

Fine Parallel Processing Using a Work Queue

Indexed Job for Parallel Processing with Static Work Assignment

Parallel Processing using Expansions

Running Automated Tasks with a CronJob

CronJobs was promoted to general availability in Kubernetes v1.21. If you are using an older version of Kubernetes, please refer to the documentation for the version of Kubernetes that you are using, so that you see accurate information. Older Kubernetes versions do not support the `batch/v1` CronJob API.

You can use a CronJob to run Jobs on a time-based schedule. These automated jobs run like Cron tasks on a Linux or UNIX system.

Cron jobs are useful for creating periodic and recurring tasks, like running backups or sending emails. Cron jobs can also schedule individual tasks for a specific time, such as if you want to schedule a job for a low activity period.

Cron jobs have limitations and idiosyncrasies. For example, in certain circumstances, a single cron job can create multiple jobs. Therefore, jobs should be idempotent.

For more limitations, see CronJobs.

Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 Cron Job

Cron jobs require a config file. This example cron job config `.spec` file prints the current time and a hello message every minute:

`application/job/cronjob.yaml`

```
apiVersion: batch/v1
kind: CronJob
```

```yaml
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Run the example CronJob by using this command:

```
kubectl create -f https://k8s.io/examples/application/job/
cronjob.yaml
```

The output is similar to this:

```
cronjob.batch/hello created
```

After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

The output is similar to this:

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        <none>          10s
```

As you can see from the results of the command, the cron job has not scheduled or run any jobs yet. Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

The output is similar to this:

```
NAME               COMPLETIONS   DURATION   AGE
hello-4111706356   0/1                      0s
hello-4111706356   0/1           0s         0s
hello-4111706356   1/1           5s         5s
```

Now you've seen one running job scheduled by the "hello" cron job. You can stop watching the job and view the cron job again to see that it scheduled the job:

```
kubectl get cronjob hello
```

The output is similar to this:

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        50s             75s
```

You should see that the cron job `hello` successfully scheduled a job at the time specified in `LAST SCHEDULE`. There are currently 0 active jobs, meaning that the job has completed or failed.

Now, find the pods that the last scheduled job created and view the standard output of one of the pods.

Note: The job name and pod name are different.

```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --
output=jsonpath={.items[*].metadata.name})
```

Show pod log:

```
kubectl logs $pods
```

The output is similar to this:

```
Fri Feb 22 11:02:09 UTC 2019
Hello from the Kubernetes cluster
```

Deleting a Cron Job

When you don't need a cron job any more, delete it with `kubectl delete cronjob <cronjob name>`:

```
kubectl delete cronjob hello
```

Deleting the cron job removes all the jobs and pods it created and stops it from creating additional jobs. You can read more about removing jobs in garbage collection.

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, and using kubectl to manage resources documents.

A cron job config also needs a `.spec` section.

Note: All modifications to a cron job, especially its `.spec`, are applied only to the following runs.

Schedule

The `.spec.schedule` is a required field of the `.spec`. It takes a Cron format string, such as `0 * * * *` or `@hourly`, as schedule time of its jobs to be created and executed.

The format also includes extended "Vixie cron" step values. As explained in the FreeBSD manual:

> Step values can be used in conjunction with ranges. Following a range with `/<number>` specifies skips of the number's value through the range. For example, `0-23/2` can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is `0,2,4,6,8,10,12,14,16,18,20,22`). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use `*/2`.

Note: A 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.

Job Template

The `.spec.jobTemplate` is the template for the job, and it is required. It has exactly the same schema as a Job, except that it is nested and does not have an `apiVersion` or `kind`. For information about writing a job `.spec`, see Writing a Job Spec.

Starting Deadline

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. After the deadline, the cron job does not start the job. Jobs that do not meet their deadline in this way count as failed jobs. If this field is not specified, the jobs have no deadline.

If the `.spec.startingDeadlineSeconds` field is set (not null), the CronJob controller measures the time between when a job is expected to be created and now. If the difference is higher than that limit, it will skip this execution.

For example, if it is set to `200`, it allows a job to be created for up to 200 seconds after the actual schedule.

Concurrency Policy

The `.spec.concurrencyPolicy` field is also optional. It specifies how to treat concurrent executions of a job that is created by this cron job. The spec may specify only one of the following concurrency policies:

- `Allow` (default): The cron job allows concurrently running jobs
- `Forbid`: The cron job does not allow concurrent runs; if it is time for a new job run and the previous job run hasn't finished yet, the cron job skips the new job run

- `Replace`: If it is time for a new job run and the previous job run hasn't finished yet, the cron job replaces the currently running job run with a new job run

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 it is set to `true`, all subsequent executions are suspended. This setting does not apply to already started executions. Defaults to false.

Caution: Executions that are suspended during their scheduled time count as missed jobs. When `.spec.suspend` changes from `true` to `false` on an existing cron job without a starting deadline, the missed jobs are scheduled immediately.

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.

Coarse Parallel Processing Using a Work Queue

In this example, we will run a Kubernetes Job with multiple parallel worker processes.

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 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.

Before you begin

Be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Starting a message queue service

This example uses RabbitMQ, however, you can adapt the example to use 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 https://raw.githubusercontent.com/kubernetes/
kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-
service.yaml
```

```
service "rabbitmq-service" created
```

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/
kubernetes/release-1.3/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:18.04
```

```
Waiting for pod default/temp-loe07 to be running, status is
Pending, pod ready: false
... [ 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 dnsutils
.... [ 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:5672

# 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` prints out the characters read from standard input, and the echo adds a carriage return so the example is readable.

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are 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:

application/job/rabbitmq/worker.py

```
#!/usr/bin/env python

# Just prints standard out and sleeps for 10 seconds.
import sys
import time
print("Processing " + sys.stdin.readlines()[0])
time.sleep(10)
```

Give the script execution permission:

```
chmod +x worker.py
```

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`.

application/job/rabbitmq/job.yaml

```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 apply -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              Message
  â”€â”€â”€â”€â”€â”€â”€â”€â”€  â”€â”€â”€â”€â”€â”€â”€â”€
â”€â”€â”€â”€â”€    â”€â”€â”€â”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€    â”€â”€â”€â”€â”€â”€
â”€â”€â”€â”€â”€â”€â”€              â”€â”€â”€â”€â”€â”€â”€
  27s        27s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-hcobb
  27s        27s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-weytj
  27s        27s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-qaam5
  27s        27s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-b67sr
  26s        26s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-xe5hj
  15s        15s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-w2zqe
  14s        14s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-d6ppa
  14s        14s        1        {job }
Normal    SuccessfulCreate    Created pod: job-wq-1-p17e0
```

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 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

In this example, we will run a Kubernetes Job with multiple parallel worker processes in a given pod.

In this example, as each pod is created, it picks up one unit of work from a task queue, processes it, and repeats until the end of the queue is reached.

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 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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to

run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Be familiar with the basic, non-parallel, use of Job.

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.

You could also download the following files directly:

- `redis-pod.yaml`
- `redis-service.yaml`
- `Dockerfile`
- `job.yaml`
- `rediswq.py`
- `worker.py`

Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are 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: false
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:

`application/job/redis/worker.py`

```python
#!/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=host)
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")
```

You could also download `worker.py`, `rediswq.py`, and `Dockerfile` files, 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:

`application/job/redis/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 apply -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        Reason            Message
  ---------    --------    -----    ----
```

```
-------------    --------    ------          -------
  33s          33s         1        {job-controller }
    Normal      SuccessfulCreate  Created pod: job-wq-2-lglf8


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 `ReplicaSet` instead, and consider running a background processing library such as https://github.com/resque/resque.

Indexed Job for Parallel Processing with Static Work Assignment

FEATURE STATE: `Kubernetes v1.22 [beta]`

In this example, you will run a Kubernetes Job that uses multiple parallel worker processes. Each worker is a different container running in its own Pod. The Pods have an *index number* that the control plane sets automatically, which allows each Pod to identify which part of the overall task to work on.

The pod index is available in the annotation `batch.kubernetes.io/job-completion-index` as a string representing its decimal value. In order for the containerized task process to obtain this index, you can publish the value of the annotation using the downward API mechanism. For convenience, the control plane automatically sets the downward API to expose the index in the `JOB_COMPLETION_INDEX` environment variable.

Here is an overview of the steps in this example:

1. **Define a Job manifest using indexed completion**. The downward API allows you to pass the pod index annotation as an environment variable or file to the container.
2. **Start an Indexed Job based on that manifest**.

Before you begin

You should already be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version v1.21. To check the version, enter `kubectl version`.

Choose an approach

To access the work item from the worker program, you have a few options:

1. Read the `JOB_COMPLETION_INDEX` environment variable. The Job controller automatically links this variable to the annotation containing the completion index.
2. Read a file that contains the completion index.
3. Assuming that you can't modify the program, you can wrap it with a script that reads the index using any of the methods above and converts it into something that the program can use as input.

For this example, imagine that you chose option 3 and you want to run the rev utility. This program accepts a file as an argument and prints its content reversed.

```
rev data.txt
```

You'll use the `rev` tool from the `busybox` container image.

As this is only an example, each Pod only does a tiny piece of work (reversing a short string). In a real workload you might, for example, create a Job that represents the task of producing 60 seconds of video based on scene data. Each work item in the video rendering Job would be to render a particular frame of that video clip. Indexed completion would mean that each Pod in the Job knows which frame to render and publish, by counting frames from the start of the clip.

Define an Indexed Job

Here is a sample Job manifest that uses `Indexed` completion mode:

`application/job/indexed-job.yaml`

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: 'indexed-job'
spec:
  completions: 5
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      restartPolicy: Never
      initContainers:
      - name: 'input'
        image: 'docker.io/library/bash'
        command:
        - "bash"
        - "-c"
        - |
          items=(foo bar baz qux xyz)
          echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt

        volumeMounts:
        - mountPath: /input
          name: input
      containers:
      - name: 'worker'
        image: 'docker.io/library/busybox'
        command:
        - "rev"
        - "/input/data.txt"
        volumeMounts:
        - mountPath: /input
          name: input
      volumes:
      - name: input
        emptyDir: {}
```

In the example above, you use the builtin `JOB_COMPLETION_INDEX` environment variable set by the Job controller for all containers. An init container maps the index to a static value and writes it to a file that is shared with the container running the worker through an emptyDir volume. Optionally, you can define your own environment variable through the downward API to publish the index to containers. You can also choose to load a list of values from a ConfigMap as an environment variable or file.

Alternatively, you can directly use the downward API to pass the annotation value as a volume file, like shown in the following example:

`application/job/indexed-job-vol.yaml`

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: 'indexed-job'
spec:
  completions: 5
  parallelism: 3
  completionMode: Indexed
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: 'worker'
        image: 'docker.io/library/busybox'
        command:
        - "rev"
        - "/input/data.txt"
        volumeMounts:
        - mountPath: /input
          name: input
      volumes:
      - name: input
        downwardAPI:
          items:
          - path: "data.txt"
            fieldRef:
              fieldPath: metadata.annotations['batch.kubernetes.i
o/job-completion-index']
```

Running the Job

Now run the Job:

```
# This uses the first approach (relying on $JOB_COMPLETION_INDEX)
kubectl apply -f https://kubernetes.io/examples/application/job/
indexed-job.yaml
```

When you create this Job, the control plane creates a series of Pods, one for each index you specified. The value of `.spec.parallelism` determines how many can run at once whereas `.spec.completions` determines how many Pods the Job creates in total.

Because `.spec.parallelism` is less than `.spec.completions`, the control plane waits for some of the first Pods to complete before starting more of them.

Once you have created the Job, wait a moment then check on progress:

```
kubectl describe jobs/indexed-job
```

The output is similar to:

```
Name:              indexed-job
Namespace:         default
Selector:          controller-
uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
Labels:            controller-
uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
                   job-name=indexed-job
Annotations:       <none>
Parallelism:       3
Completions:       5
Start Time:        Thu, 11 Mar 2021 15:47:34 +0000
Pods Statuses:     2 Running / 3 Succeeded / 0 Failed
Completed Indexes: 0-2
Pod Template:
  Labels:  controller-uid=bf865e04-0b67-483b-9a90-74cfc4c3e756
           job-name=indexed-job
  Init Containers:
   input:
    Image:      docker.io/library/bash
    Port:       <none>
    Host Port:  <none>
    Command:
      bash
      -c
      items=(foo bar baz qux xyz)
      echo ${items[$JOB_COMPLETION_INDEX]} > /input/data.txt

    Environment:  <none>
    Mounts:
      /input from input (rw)
  Containers:
   worker:
    Image:      docker.io/library/busybox
    Port:       <none>
    Host Port:  <none>
    Command:
      rev
      /input/data.txt
    Environment:  <none>
    Mounts:
      /input from input (rw)
  Volumes:
   input:
    Type:       EmptyDir (a temporary directory that shares a
pod's lifetime)
    Medium:
    SizeLimit:  <unset>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  4s    job-controller  Created pod:
indexed-job-njkjj
```

```
  Normal  SuccessfulCreate  4s    job-controller  Created pod:
indexed-job-9kd4h
  Normal  SuccessfulCreate  4s    job-controller  Created pod:
indexed-job-qjwsz
  Normal  SuccessfulCreate  1s    job-controller  Created pod:
indexed-job-fdhq5
  Normal  SuccessfulCreate  1s    job-controller  Created pod:
indexed-job-ncslj
```

In this example, you run the Job with custom values for each index. You can inspect the output of one of the pods:

```
kubectl logs indexed-job-fdhq5 # Change this to match the name
of a Pod from that Job
```

The output is similar to:

```
xuq
```

Parallel Processing using Expansions

This task demonstrates running multiple Jobs based on a common template. You can use this approach to process batches of work in parallel.

For this example there are only three items: *apple*, *banana*, and *cherry*. The sample Jobs process each item by printing a string then pausing.

See using Jobs in real workloads to learn about how this pattern fits more realistic use cases.

Before you begin

You should be familiar with the basic, non-parallel, use of Job.

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

For basic templating you need the command-line utility `sed`.

To follow the advanced templating example, you need a working installation of Python, and the Jinja2 template library for Python.

Once you have Python set up, you can install Jinja2 by running:

```
pip install --user jinja2
```

Create Jobs based on a template

First, download the following template of a Job to a file called `job-tmpl.yaml`. Here's what you'll download:

application/job/job-tmpl.yaml

```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:1.28
        command: ["sh", "-c", "echo Processing item $ITEM &&
sleep 5"]
      restartPolicy: Never
```

```
# Use curl to download job-tmpl.yaml
curl -L -s -O https://k8s.io/examples/application/job/job-
tmpl.yaml
```

The file you downloaded is not yet a valid Kubernetes manifest. Instead that template is a YAML representation of a Job object with some placeholders that need to be filled in before it can be used. The `$ITEM` syntax is not meaningful to Kubernetes.

Create manifests from the template

The following shell snippet uses `sed` to replace the string `$ITEM` with the loop variable, writing into a temporary directory named `jobs`. Run this now:

```shell
# Expand the template into multiple files, one for each item to
be processed.
mkdir ./jobs
for i in apple banana cherry
do
  cat job-tmpl.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done
```

Check if it worked:

```
ls jobs/
```

The output is similar to this:

```
job-apple.yaml
job-banana.yaml
job-cherry.yaml
```

You could use any type of template language (for example: Jinja2; ERB), or write a program to generate the Job manifests.

Create Jobs from the manifests

Next, create all the Jobs with one kubectl command:

```
kubectl create -f ./jobs
```

The output is similar to this:

```
job.batch/process-item-apple created
job.batch/process-item-banana created
job.batch/process-item-cherry created
```

Now, check on the jobs:

```
kubectl get jobs -l jobgroup=jobexample
```

The output is similar to this:

```
NAME                    COMPLETIONS   DURATION   AGE
process-item-apple      1/1           14s        22s
process-item-banana     1/1           12s        21s
process-item-cherry     1/1           12s        20s
```

Using the `-l` option to kubectl selects only the Jobs that are part of this group of jobs (there might be other unrelated jobs in the system).

You can check on the Pods as well using the same label selector:

```
kubectl get pods -l jobgroup=jobexample
```

The output is similar to:

```
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
```

We can use this single command to check on the output of all jobs at once:

```
kubectl logs -f -l jobgroup=jobexample
```

The output should be:

```
Processing item apple
Processing item banana
Processing item cherry
```

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Use advanced template parameters

In the first example, each instance of the template had one parameter, and that parameter was also used in the Job's name. However, names are restricted to contain only certain characters.

This slightly more complex example uses the Jinja template language to generate manifests and then objects from those manifests, with a multiple parameters for each Job.

For this part of the task, you are going to use a one-line Python script to convert the template to a set of manifests.

First, copy and paste the following template of a Job object, into a file called `job.yaml.jinja2`:

```
{% set params = [{ "name": "apple", "url": "http://dbpedia.org/
resource/Apple", },
                  { "name": "banana", "url": "http://dbpedia.org/
resource/Banana", },
                  { "name": "cherry", "url": "http://dbpedia.org/
resource/Cherry" }]
%}
{% 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:1.28
        command: ["sh", "-c", "echo Processing URL {{ url }} &&
sleep 5"]
      restartPolicy: Never
{% endfor %}
```

The above template defines two parameters for each Job object using a list of python dicts (lines 1-4). A `for` loop emits one Job manifest for each set of parameters (remaining lines).

This example relies on a feature of YAML. One YAML file can contain multiple documents (Kubernetes manifests, in this case), separated by `---` on a line by itself. You can pipe the output directly to `kubectl` to create the Jobs.

Next, use this one-line Python program to expand the template:

```
alias render_template='python -c "from jinja2 import Template;
import sys; print(Template(sys.stdin.read()).render());"'
```

Use `render_template` to convert the parameters and template into a single YAML file containing Kubernetes manifests:

```
# This requires the alias you defined earlier
cat job.yaml.jinja2 | render_template > jobs.yaml
```

You can view `jobs.yaml` to verify that the `render_template` script worked correctly.

Once you are happy that `render_template` is working how you intend, you can pipe its output into `kubectl`:

```
cat job.yaml.jinja2 | render_template | kubectl apply -f -
```

Kubernetes accepts and runs the Jobs you created.

Clean up

```
# Remove the Jobs you created
# Your cluster automatically cleans up their Pods
kubectl delete job -l jobgroup=jobexample
```

Using Jobs in real workloads

In a real use case, each Job performs some substantial computation, such as rendering a frame of a movie, or processing a range of rows in a database. If you were rendering a movie you would set `$ITEM` to the frame number. If you were processing rows from a database table, you would set `$ITEM` to represent the range of database rows to process.

In the task, you ran a command to collect the output from Pods by fetching their logs. In a real use case, each Pod for a Job writes its output to durable storage before completing. You can use a PersistentVolume for each Job, or an external storage service. For example, if you are rendering frames for a movie, use HTTP to `PUT` the rendered frame data to a URL, using a different URL for each frame.

Labels on Jobs and Pods

After you create a Job, Kubernetes automatically adds additional labels that distinguish one Job's pods from another Job's pods.

In this example, each Job and its Pod template have a label: `jobgroup=jobexample`.

Kubernetes itself pays no attention to labels named `jobgroup`. Setting a label for all the Jobs you create from a template makes it convenient to operate on all those Jobs at once. In the first example you used a template to create several Jobs. The template ensures that each Pod also gets the same label, so you can check on all Pods for these templated Jobs with a single command.

Note: The label key `jobgroup` is not special or reserved. You can pick your own labelling scheme. There are recommended labels that you can use if you wish.

Alternatives

If you plan to create a large number of Job objects, you may find that:

- Even using labels, managing so many Jobs is cumbersome.
- If you create many Jobs in a batch, you might place high load on the Kubernetes control plane. Alternatively, the Kubernetes API server could rate limit you, temporarily rejecting your requests with a 429 status.
- You are limited by a resource quota on Jobs: the API server permanently rejects some of your requests when you create a great deal of work in one batch.

There are other job patterns that you can use to process large amounts of work without creating very many Job objects.

You could also consider writing your own controller to manage Job objects automatically.

Access Applications in a Cluster

Configure load balancing, port forwarding, or setup firewall or DNS configurations to access applications in a cluster.

Deploy and Access the Kubernetes Dashboard

Deploy the web UI (Kubernetes Dashboard) and access it.

Accessing Clusters

Configure Access to Multiple Clusters

Use Port Forwarding to Access Applications in a Cluster

Use a Service to Access an Application in a Cluster

Connect a Frontend to a Backend Using Services

Create an External Load Balancer

List All Container Images Running in a Cluster

Set up Ingress on Minikube with the NGINX Ingress Controller

Communicate Between Containers in the Same Pod Using a Shared Volume

Configure DNS for a Cluster

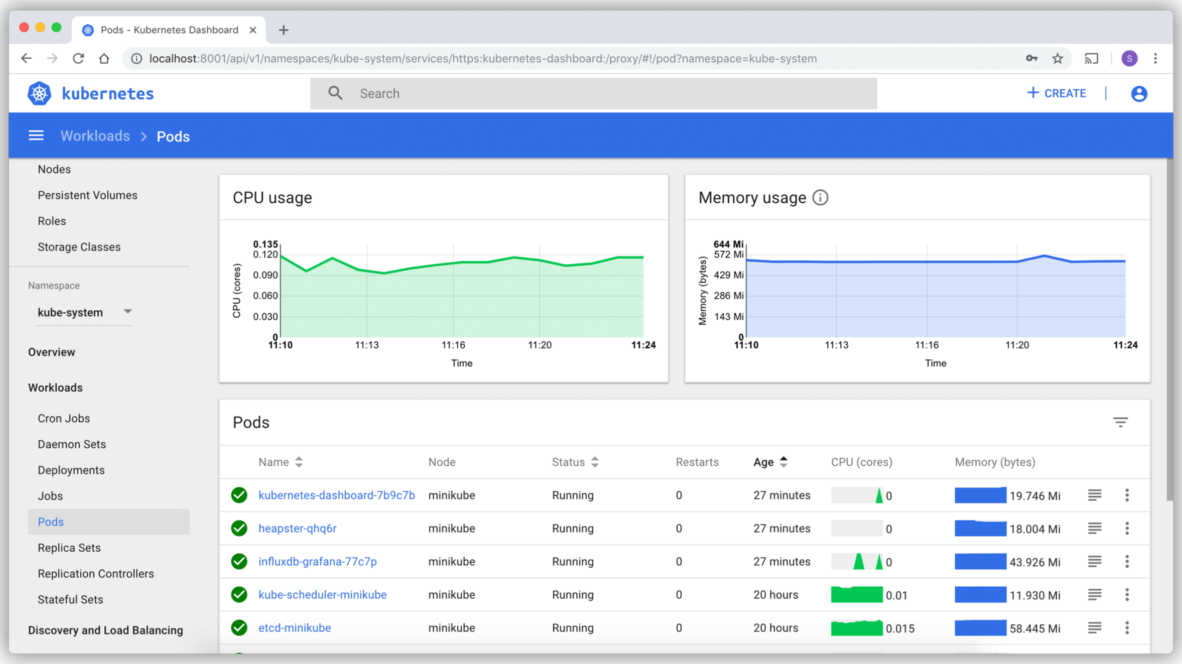
Access Services Running on Clusters

Deploy and Access the Kubernetes Dashboard

Deploy the web UI (Kubernetes Dashboard) and access it.

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 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	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
kubernetes-dashboard-7b9c7b	minikube	Running	0	27 minutes	0	19.746 Mi
heapster-qhq6r	minikube	Running	0	27 minutes	0	18.004 Mi
influxdb-grafana-77c7p	minikube	Running	0	27 minutes	0	43.926 Mi
kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

Deploying the Dashboard UI

The Dashboard UI is not deployed by default. To deploy it, run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.5.0/aio/deploy/recommended.yaml
```

Accessing the Dashboard UI

To protect your cluster data, Dashboard deploys with a minimal RBAC configuration by default. Currently, Dashboard only supports logging in with a Bearer Token. To create a token for this demo, you can follow our guide on creating a sample user.

Warning: The sample user created in the tutorial will have administrative privileges and is for educational purposes only.

Command line proxy

You can enable access to the Dashboard using the `kubectl` command-line tool, by running the following command:

```
kubectl proxy
```

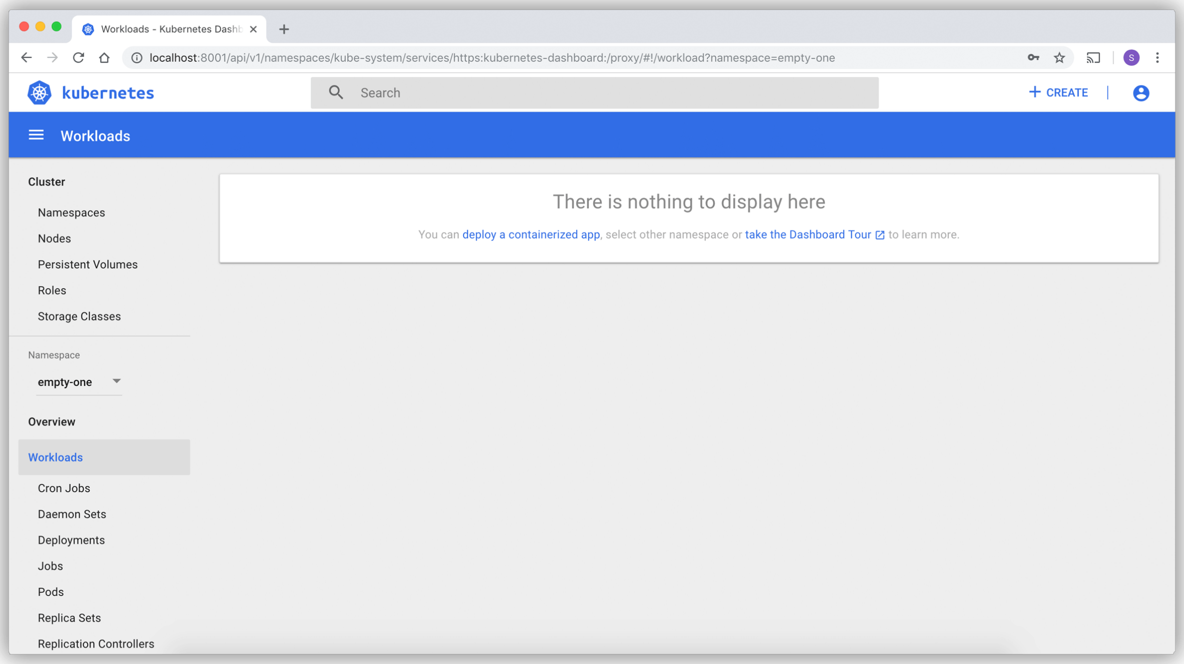
Kubectl will make Dashboard available at http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/.

The UI can *only* be accessed from the machine where the command is executed. See `kubectl proxy --help` for more options.

Note: The kubeconfig authentication method does **not** support external identity providers or X.509 certificate-based authentication.

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 *manifest* file containing application configuration.

Click the **CREATE** button in the upper right corner of any page to begin.

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).

  Note: For external Services, you may need to open up one or more ports to do so.

  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, for example `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 manifests (YAML or JSON configuration files). The manifests use Kubernetes API resource schemas.

As an alternative to specifying application details in the deploy wizard, you can define your application in one or more manifests, and upload the files using Dashboard.

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 overview

For cluster and namespace administrators, Dashboard lists Nodes, Namespaces and PersistentVolumes 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.

Workloads

Shows all applications running in the selected namespace. The view lists applications by workload kind (for example: Deployments, ReplicaSets, StatefulSets). Each workload kind can be viewed separately. The lists summarize actionable information about the workloads, such as the number of ready pods for a ReplicaSet or current memory usage for a Pod.

Detail views for workloads show status and specification information and surface relationships between objects. For example, Pods that ReplicaSet is controlling or new ReplicaSets and HorizontalPodAutoscalers for Deployments.

Services

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.

Storage

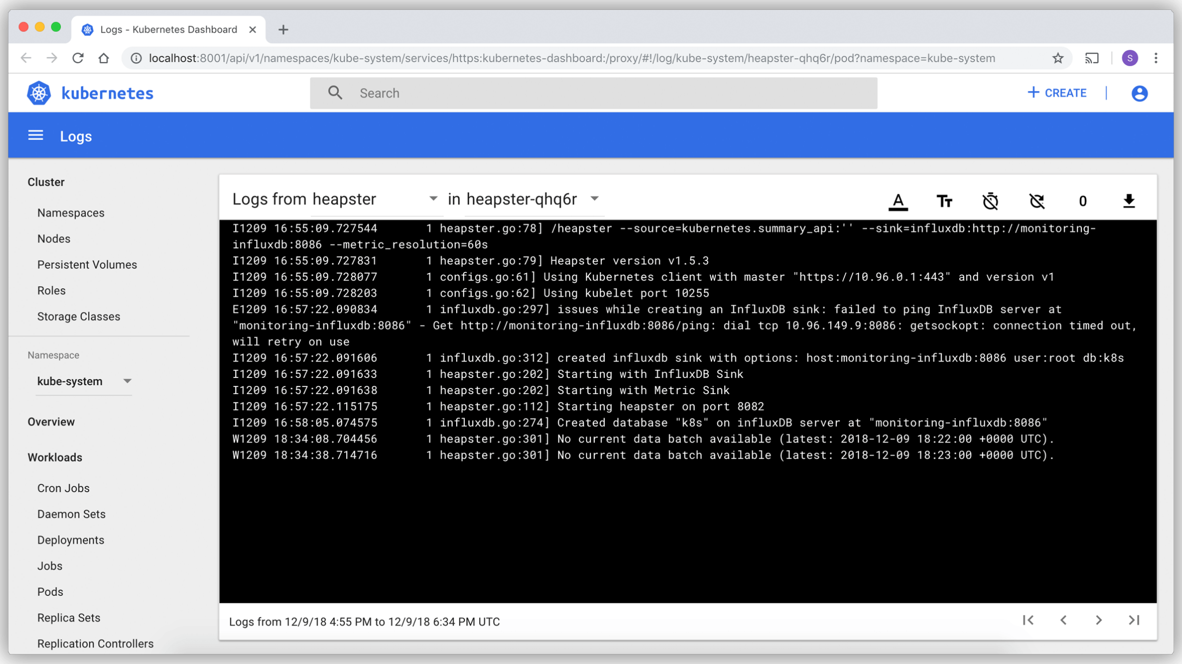
Storage view shows PersistentVolumeClaim resources which are used by applications for storing data.

ConfigMaps and Secrets

Shows all Kubernetes resources that are used for live configuration of applications running in clusters. The view allows for editing and managing config objects and displays secrets hidden by default.

Logs viewer

Pod lists and detail pages link to a logs viewer that is built into Dashboard. The viewer allows for drilling down logs from containers belonging to a single Pod.



```
I1209 16:55:09.727544       1 heapster.go:78] /heapster --source=kubernetes.summary_api:'' --sink=influxdb:http://monitoring-
influxdb:8086 --metric_resolution=60s
I1209 16:55:09.727831       1 heapster.go:79] Heapster version v1.5.3
I1209 16:55:09.728077       1 configs.go:61] Using Kubernetes client with master "https://10.96.0.1:443" and version v1
I1209 16:55:09.728203       1 configs.go:62] Using kubelet port 10255
E1209 16:57:22.090834       1 influxdb.go:297] issues while creating an InfluxDB sink: failed to ping InfluxDB server at
"monitoring-influxdb:8086" - Get http://monitoring-influxdb:8086/ping: dial tcp 10.96.149.9:8086: getsockopt: connection timed out,
will retry on use
I1209 16:57:22.091606       1 influxdb.go:312] created influxdb sink with options: host:monitoring-influxdb:8086 user:root db:k8s
I1209 16:57:22.091633       1 heapster.go:202] Starting with InfluxDB Sink
I1209 16:57:22.091638       1 heapster.go:202] Starting with Metric Sink
I1209 16:57:22.115175       1 heapster.go:112] Starting heapster on port 8082
I1209 16:58:05.074575       1 influxdb.go:274] Created database "k8s" on influxDB server at "monitoring-influxdb:8086"
W1209 18:34:08.704456       1 heapster.go:301] No current data batch available (latest: 2018-12-09 18:22:00 +0000 UTC).
W1209 18:34:38.714716       1 heapster.go:301] No current data batch available (latest: 2018-12-09 18:23:00 +0000 UTC).
```

Logs from 12/9/18 4:55 PM to 12/9/18 6:34 PM UTC

What's next

For more information, see the [Kubernetes Dashboard project page]().

Accessing Clusters

This topic discusses multiple ways to interact with clusters.

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 reference.

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, replacing localhost with [::1] for IPv6, like so:

```
curl http://localhost:8080/api/
```

The output is similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

Without kubectl proxy

Use `kubectl apply` and `kubectl describe secret...` to create a token for the default service account with grep/cut:

First, create the Secret, requesting a token for the default ServiceAccount:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: default-token
  annotations:
    kubernetes.io/service-account.name: default
type: kubernetes.io/service-account-token
EOF
```

Next, wait for the token controller to populate the Secret with a token:

```
while ! kubectl describe secret default-token | grep -E '^token'
>/dev/null; do
  echo "waiting for token..." >&2
  sleep 1
done
```

Capture and use the generated token:

```
APISERVER=$(kubectl config view --minify | grep server | cut -f
2- -d ":" | tr -d " ")
TOKEN=$(kubectl describe secret default-token | grep -E '^token'
| cut -f2 -d':' | tr -d " ")

curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --
insecure
```

The output is similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

Using `jsonpath`:

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
TOKEN=$(kubectl get secret default-token -o jsonpath='{.data.token}' | base64 --decode)

curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

The output is similar to this:

```
{
  "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. Controlling Access to the API describes how a cluster admin can configure this.

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@kubernetes-<kubernetes-version-number>`, see INSTALL.md for detailed installation instructions. 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/kubernetes"` 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 API server are somewhat different.

Please check Accessing the API from within a Pod for more details.

Accessing services running on the cluster

The previous section describes how to connect to the Kubernetes API server. For information about connecting to other services running on a Kubernetes cluster, see Access Cluster Services.

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 only 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`.
Warning: Only use kubeconfig files from trusted sources. Using a specially-crafted kubeconfig file could result in malicious code execution or file exposure. If you must use an untrusted kubeconfig file, inspect it carefully first, much as you would a shell script.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check that kubectl is installed, run `kubectl version --client`. The kubectl version should be within one minor version of your cluster's API server.

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://1.2.3.4 --certificate-authority=fake-ca-file
kubectl config --kubeconfig=config-demo set-cluster scratch --
server=https://5.6.7.8 --insecure-skip-tls-verify
```

Add user details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-credentials
developer --client-certificate=fake-cert-file --client-key=fake-
key-seefile
kubectl config --kubeconfig=config-demo set-credentials
experimenter --username=exp --password=some-password
```

Note:

- To delete a user you can run `kubectl --kubeconfig=config-demo config unset users.<name>`
- To remove a cluster, you can run `kubectl --kubeconfig=config-demo config unset clusters.<name>`
- To remove a context, you can run `kubectl --kubeconfig=config-demo config unset contexts.<name>`

Add context details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-context dev-frontend
--cluster=development --namespace=frontend --user=developer
kubectl config --kubeconfig=config-demo set-context dev-storage
--cluster=development --namespace=storage --user=developer
```

```
kubectl config --kubeconfig=config-demo set-context exp-scratch
--cluster=scratch --namespace=default --user=experimenter
```

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
```

The `fake-ca-file`, `fake-cert-file` and `fake-key-file` above are the placeholders for the pathnames of the certificate files. You need to change these to the actual pathnames of certificate files in your environment.

Sometimes you may want to use Base64-encoded data embedded here instead of separate certificate files; in that case you need to add the suffix `-data` to the keys, for example, `certificate-authority-data`, `client-certificate-data`, `client-key-data`.

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 `exp-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:

Linux

```
export KUBECONFIG_SAVED=$KUBECONFIG
```

Windows PowerShell

```
$Env:KUBECONFIG_SAVED=$ENV: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:

Linux

```
export KUBECONFIG=$KUBECONFIG:config-demo:config-demo-2
```

Windows PowerShell

```
$Env: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:

Linux

```
export KUBECONFIG=$KUBECONFIG:$HOME/.kube/config
```

Windows Powershell

```
$Env:KUBECONFIG="$Env: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:

Linux

```
export KUBECONFIG=$KUBECONFIG_SAVED
```

Windows PowerShell

```
$Env:KUBECONFIG=$ENV: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 MongoDB 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

  Your Kubernetes server must be at or later than version v1.10. To check the version, enter `kubectl version`.
- Install MongoDB Shell.

Creating MongoDB deployment and service

1. Create a Deployment that runs MongoDB:

   ```
   kubectl apply -f https://k8s.io/examples/application/mongodb/
   mongo-deployment.yaml
   ```

   The output of a successful command verifies that the deployment was created:

   ```
   deployment.apps/mongo created
   ```

   View the pod status to check that it is ready:

   ```
   kubectl get pods
   ```

   The output displays the pod created:

   ```
   NAME                       READY   STATUS    RESTARTS   AGE
   mongo-75f59d57f4-4nd6q     1/1     Running   0          2m4s
   ```

   View the Deployment's status:

   ```
   kubectl get deployment
   ```

   The output displays that the Deployment was created:

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
mongo   1/1     1            1           2m21s
```

The Deployment automatically manages a ReplicaSet. View the ReplicaSet status using:

```
kubectl get replicaset
```

The output displays that the ReplicaSet was created:

```
NAME               DESIRED   CURRENT   READY   AGE
mongo-75f59d57f4   1         1         1       3m12s
```

2. Create a Service to expose MongoDB on the network:

```
kubectl apply -f https://k8s.io/examples/application/mongodb/
mongo-service.yaml
```

The output of a successful command verifies that the Service was created:

```
service/mongo created
```

Check the Service created:

```
kubectl get service mongo
```

The output displays the service created:

```
NAME    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
mongo   ClusterIP   10.96.41.183   <none>        27017/TCP
11s
```

3. Verify that the MongoDB server is running in the Pod, and listening on port 27017:

```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod
kubectl get pod mongo-75f59d57f4-4nd6q --template='{{(index
(index .spec.containers 0).ports 0).containerPort}}{{"\n"}}'
```

The output displays the port for MongoDB in that Pod:

```
27017
```

(this is the TCP port allocated to MongoDB on the internet).

Forward a local port to a port on the Pod

1. `kubectl port-forward` allows using resource name, such as a pod name, to select a matching pod to port forward to.

```
# Change mongo-75f59d57f4-4nd6q to the name of the Pod
kubectl port-forward mongo-75f59d57f4-4nd6q 28015:27017
```

which is the same as

```
kubectl port-forward pods/mongo-75f59d57f4-4nd6q 28015:27017
```

or

```
kubectl port-forward deployment/mongo 28015:27017
```

or

```
kubectl port-forward replicaset/mongo-75f59d57f4 28015:27017
```

or

```
kubectl port-forward service/mongo 28015:27017
```

Any of the above commands works. The output is similar to this:

```
Forwarding from 127.0.0.1:28015 -> 27017
Forwarding from [::1]:28015 -> 27017
```

Note: `kubectl port-forward` does not return. To continue with the exercises, you will need to open another terminal.

1. Start the MongoDB command line interface:

   ```
   mongosh --port 28015
   ```

2. At the MongoDB command line prompt, enter the `ping` command:

   ```
   db.runCommand( { ping: 1 } )
   ```

   A successful ping request returns:

   ```
   { ok: 1 }
   ```

Optionally let *kubectl* choose the local port

If you don't need a specific local port, you can let `kubectl` choose and allocate the local port and thus relieve you from having to manage local port conflicts, with the slightly simpler syntax:

```
kubectl port-forward deployment/mongo :27017
```

The `kubectl` tool finds a local port number that is not in use (avoiding low ports numbers, because these might be used by other applications). The output is similar to:

```
Forwarding from 127.0.0.1:63753 -> 27017
Forwarding from [::1]:63753 -> 27017
```

Discussion

Connections made to local port 28015 are forwarded to port 27017 of the Pod that is running the MongoDB server. With this connection in place, you can use your local workstation to debug the database that is running in the Pod.

Note: `kubectl port-forward` is implemented for TCP ports only. The support for UDP protocol is tracked in issue 47862.

What's next

Learn more about kubectl port-forward.

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.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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.

Creating a service for an application running in two pods

Here is the configuration file for the application Deployment:

service/access/hello-application.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
      labels:
        run: load-balancer-example
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
              protocol: TCP
```

1. Run a Hello World application in your cluster: Create the application Deployment using the file above:

   ```
   kubectl apply -f https://k8s.io/examples/service/access/
   hello-application.yaml
   ```

   The preceding command creates a Deployment and an associated ReplicaSet. 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
TargetPort:             8080/TCP
NodePort:               <unset> 31496/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.

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.

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.

Connect a Frontend to a Backend Using Services

This task shows how to create a *frontend* and a *backend* microservice. The backend microservice is a hello greeter. The frontend exposes the backend using nginx and a Kubernetes Service object.

Objectives

- Create and run a sample `hello` backend microservice using a Deployment object.
- Use a Service object to send traffic to the backend microservice's multiple replicas.
- Create and run a `nginx` frontend microservice, also using a Deployment object.
- Configure the frontend microservice to send traffic to the backend microservice.
- Use a Service object of `type=LoadBalancer` to expose the frontend microservice outside the 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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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:

service/access/backend-deployment.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  selector:
    matchLabels:
      app: hello
      tier: backend
      track: stable
  replicas: 3
  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 apply -f https://k8s.io/examples/service/access/backend-
deployment.yaml
```

View information about the backend Deployment:

```
kubectl describe deployment backend
```

The output is similar to this:

```
Name:                           backend
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:                       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=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 (3/3 replicas
created)
Events:
...
```

Creating the `hello` Service object

The key to sending requests from 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 selectors to find the Pods that it routes traffic to.

First, explore the Service configuration file:

[service/access/backend-service.yaml]

```yaml
---
apiVersion: v1
kind: Service
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, named `hello` routes traffic to Pods that have the labels `app: hello` and `tier: backend`.

Create the backend Service:

```
kubectl apply -f https://k8s.io/examples/service/access/backend-service.yaml
```

At this point, you have a `backend` Deployment running three replicas of your `hello` application, and you have a Service that can route traffic to them. However, this service is neither available nor resolvable outside the cluster.

Creating the frontend

Now that you have your backend running, you can create a frontend that is accessible outside the cluster, and connects to the backend by proxying requests to it.

The frontend sends requests 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 `examples/service/access/backend-service.yaml` configuration file.

The Pods in the frontend Deployment run a nginx image that is configured to proxy requests to the `hello` backend Service. Here is the nginx configuration file:

[service/access/frontend-nginx.conf]

```
# The identifier Backend is internal to nginx, and used to name
this specific upstream
upstream Backend {
    # hello is the internal DNS name used by the backend Service
inside Kubernetes
    server hello;
}

server {
    listen 80;

    location / {
        # The following statement will proxy traffic to the
upstream named Backend
        proxy_pass http://Backend;
    }
}
```

Similar to the backend, the frontend has a Deployment and a Service. An important difference to notice between the backend and frontend services, is that the configuration for the frontend Service has `type: LoadBalancer`, which means that the Service uses a load balancer provisioned by your cloud provider and will be accessible from outside the cluster.

service/access/frontend-service.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: hello
    tier: frontend
  ports:
  - protocol: "TCP"
    port: 80
    targetPort: 80
  type: LoadBalancer
...
```

service/access/frontend-deployment.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
```

```
spec:
  selector:
    matchLabels:
      app: hello
      tier: frontend
      track: stable
  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 apply -f https://k8s.io/examples/service/access/frontend-
deployment.yaml
kubectl apply -f https://k8s.io/examples/service/access/frontend-
service.yaml
```

The output verifies that both resources were created:

```
deployment.apps/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 --watch
```

This displays the configuration for the `frontend` Service and watches for changes. Initially, the external IP is listed as `<pending>`:

```
NAME        TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
```

```
frontend   LoadBalancer   10.51.252.116   <pending>     80/TCP
10s
```

As soon as an external IP is provisioned, however, the configuration updates to include the new IP under the `EXTERNAL-IP` heading:

```
NAME       TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)  AGE
frontend   LoadBalancer   10.51.252.116   XXX.XXX.XXX.XXX    80/
TCP   1m
```

That IP can now be used to interact with the `frontend` service from outside the cluster.

Send traffic through the frontend

The frontend and backend 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} # replace this with the EXTERNAL-IP
you saw earlier
```

The output shows the message generated by the backend:

```
{"message":"Hello"}
```

Cleaning up

To delete the Services, enter this command:

```
kubectl delete services frontend backend
```

To delete the Deployments, the ReplicaSets and the Pods that are running the backend and frontend applications, enter this command:

```
kubectl delete deployment frontend backend
```

What's next

- Learn more about Services
- Learn more about ConfigMaps
- Learn more about DNS for Service and Pods

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 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.*

You can also use an Ingress in place of Service. For more information, check the Ingress documentation.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your cluster must be running in a cloud or other environment that already has support for configuring external load balancers.

Create a Service

Create a Service from a manifest

To create an external load balancer, add the following line to your Service manifest:

```
    type: LoadBalancer
```

Your manifest might then look like:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  type: LoadBalancer
```

Create a Service using kubectl

You can alternatively create the service with the `kubectl expose` command and its `--type=LoadBalancer` flag:

```
kubectl expose deployment 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 Deployment 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 similar to:

```
Name:                     example-service
Namespace:                default
Labels:                   app=example
Annotations:              <none>
Selector:                 app=example
Type:                     LoadBalancer
IP Families:              <none>
IP:                       10.3.22.96
IPs:                      10.3.22.96
LoadBalancer Ingress:     192.0.2.89
Port:                     <unset>  8765/TCP
TargetPort:               9376/TCP
NodePort:                 <unset>  30593/TCP
Endpoints:                172.17.0.3:9376
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

The load balancer's IP address is listed next to `LoadBalancer Ingress`.

Note:

If you are running your service on Minikube, you can find the assigned IP address and port with:

```
minikube service example-service --url
```

Preserving the client source IP

By default, the source IP seen in the target container is *not the original source IP* of the client. To enable preservation of the client IP, the following fields can be configured in the `.spec` of the 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.
- `.spec.healthCheckNodePort` - specifies the health check node port (numeric port number) for the service. If you don't specify `healthCheckNodePort`, the service controller allocates a port from your cluster's NodePort range.
  You can configure that range by setting an API server command line option, `--service-node-port-range`. The Service will use the user-specified `healthCheckNodePort` value if you specify it, provided that the Service `type` is set to LoadBalancer and `externalTrafficPolicy` is set to `Local`.

Setting `externalTrafficPolicy` to Local in the Service manifest activates this feature. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  externalTrafficPolicy: Local
  type: LoadBalancer
```

Caveats and limitations when preserving source IPs

Load balancing services from some cloud providers do not let you configure different weights for each target.

With each target weighted equally in terms of sending traffic to Nodes, external traffic is not equally load balanced across different Pods. The external load balancer is unaware of the number of Pods on each node that are used as a target.

Where `NumServicePods << _NumNodes` or `NumServicePods >> NumNodes`, a fairly close-to-equal distribution will be seen, even without weights.

Internal pod to pod traffic should behave similar to ClusterIP services, with equal probability across all pods.

Garbage collecting load balancers

FEATURE STATE: `Kubernetes v1.17 [stable]`

In usual case, the correlating load balancer resources in cloud provider should be cleaned up soon after a LoadBalancer type Service is deleted. But it is known that there are various corner cases where cloud resources are orphaned after the associated Service is deleted. Finalizer Protection for Service LoadBalancers was introduced to prevent this from happening. By using finalizers, a Service resource will never be deleted until the correlating load balancer resources are also deleted.

Specifically, if a Service has `type` LoadBalancer, the service controller will attach a finalizer named `service.kubernetes.io/load-balancer-cleanup`. The finalizer will only be removed after the load balancer resource is cleaned up. This prevents dangling load balancer resources even in corner cases such as the service controller crashing.

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` equals ClusterIP to pods within the cluster and extends it by programming the (external to Kubernetes) load balancer with entries for the nodes hosting the relevant Kubernetes pods. The Kubernetes control plane automates the creation of the external load balancer, health checks (if needed), and packet filtering rules (if needed). Once the cloud provider allocates an IP address for the load balancer, the control plane looks up that external IP address and populates it into the Service object.

What's next

- Read about Service
- Read about Ingress
- Read Connecting Applications with Services

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. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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 check the version, enter `kubectl version`.

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 Container images 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={.items[*].spec.containers[*].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="{.items[*].spec.containers[*].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, for example `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 Container images 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.name}{":\t"}{range .spec.containers[*]}{.image}
{", "}{end}{end}' |\
sort
```

List Container images 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="{.items[*].spec.co
ntainers[*].image}" -l app=nginx
```

List Container images 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="{.items[*].
spec.containers[*].image}"
```

List Container images 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="{{ra
nge .items}}{{range .spec.containers}}{{.image}} {{end}}{{end}}"
```

What's next

Reference

- Jsonpath reference guide
- Go template reference guide

Set up Ingress on Minikube with the NGINX Ingress Controller

An Ingress is an API object that defines rules which allow external access to services in a cluster. An Ingress controller fulfills the rules set in the Ingress.

This page shows you how to set up a simple Ingress which routes requests to Service web or web2 depending on the HTTP URI.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. 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

Your Kubernetes server must be at or later than version 1.19. To check the version, enter `kubectl version`. If you are using an older Kubernetes version, switch to the documentation for that version.

Create a Minikube cluster

Using Katacoda