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:

- iximiuz Labs
- Killercoda
- KodeKloud
- 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  Copy pods/inject/secret.yaml to clipboard

```
apiVersion: v1
kind: Secretmetadata:  name: test-secretdata:  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='username=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  Copy pods/inject/secret-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: secret-test-podspec:  containers:    - name: test-container      image: nginx      volumeMounts:
```

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
   ```

Modify your image or command line so that the program looks for files in the `mountPath` directory. Each key in the Secret `data` map becomes a file name in this directory.

Project Secret keys to specific file paths

You can also control the paths within the volume where Secret keys are projected. Use the `.spec.volumes[].secret.items` field to change the target path of each key:

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  containers:  - name: mypod    image: redis    volumeMounts:    - name: foo      mountPath: "
```

When you deploy this Pod, the following happens:

- The `username` key from `mysecret` is available to the container at the path `/etc/foo/my-group/my-username` instead of at `/etc/foo/username`.
- The `password` key from that Secret object is not projected.

If you list keys explicitly using `.spec.volumes[].secret.items`, consider the following:

- Only keys specified in `items` are projected.
- To consume all keys from the Secret, all of them must be listed in the `items` field.
- All listed keys must exist in the corresponding Secret. Otherwise, the volume is not created.

Set POSIX permissions for Secret keys

You can set the POSIX file access permission bits for a single Secret key. If you don't specify any permissions, `0644` is used by default. You can also set a default POSIX file mode for the entire Secret volume, and you can override per key if needed.

For example, you can specify a default mode like this:

```
apiVersion: v1
kind: Podmetadata:  name: mypodspec:  containers:  - name: mypod    image: redis    volumeMounts:    - name: foo      mountPath: "
```

The Secret is mounted on `/etc/foo`; all the files created by the secret volume mount have permission `0400`.

Note:

If you're defining a Pod or a Pod template using JSON, beware that the JSON specification doesn't support octal literals for numbers because JSON considers `0400` to be the *decimal* value `400`. In JSON, use decimal values for the `defaultMode` instead. If you're writing YAML, you can write the `defaultMode` in octal.

Define container environment variables using Secret data

You can consume the data in Secrets as environment variables in your containers.

If a container already consumes a Secret in an environment variable, a Secret update will not be seen by the container unless it is restarted. There are third party solutions for triggering restarts when secrets change.

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=backend-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  Copy pods/inject/pod-single-secret-env-variable.yaml to clipboard

  ```
  apiVersion: v1
  kind: Podmetadata:  name: env-single-secretspec:  containers:  - name: envars-test-container    image: nginx    env:    - nam
  ```

- 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 similar to:

  ```
  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 Copy pods/inject/pod-multiple-secret-env-variable.yaml to clipboard

  ```
  apiVersion: v1
  kind: Podmetadata:  name: envvars-multiple-secretsspec:  containers:  - name: envars-test-container    image: nginx    env:
  ```

- 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 'env | grep _USERNAME'
  ```

  The output is similar to:

  ```
  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=username='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 Copy pods/inject/pod-secret-envFrom.yaml to clipboard

  ```
  apiVersion: v1
  kind: Podmetadata:  name: envfrom-secretspec:  containers:  - name: envars-test-container    image: nginx    envFrom:    - se
  ```

- 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 similar to:

  ```
  username: my-app
  password: 39528$vdg7Jb
  ```

Example: Provide prod/test credentials to Pods using Secrets

This example illustrates a Pod which consumes a secret containing production credentials and another Pod which consumes a secret with test environment credentials.

1. Create a secret for prod environment credentials:

   ```
   kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11
   ```

   The output is similar to:

   ```
   secret "prod-db-secret" created
   ```

2. Create a secret for test environment credentials.

   ```
   kubectl create secret generic test-db-secret --from-literal=username=testuser --from-literal=password=iluvtests
   ```

   The output is similar to:

   ```
   secret "test-db-secret" created
   ```

   Note:

   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 actual password is `S!B\*d$zDsb=`, you should execute the command as follows:

   ```
   kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-literal=password='S!B\*d$zDsb='
   ```

   You do not need to escape special characters in passwords from files (`--from-file`).

3. Create the Pod manifests:

   ```
   cat <<EOF > pod.yaml
   apiVersion: v1
   kind: List
   items:
   - kind: Pod
     apiVersion: v1
     metadata:
       name: prod-db-client-pod
       labels:
         name: prod-db-client
     spec:
       volumes:
       - name: secret-volume
         secret:
           secretName: prod-db-secret
       containers:
       - name: db-client-container
         image: myClientImage
         volumeMounts:
         - name: secret-volume
           readOnly: true
           mountPath: "/etc/secret-volume"
   - kind: Pod
     apiVersion: v1
     metadata:
       name: test-db-client-pod
       labels:
         name: test-db-client
     spec:
       volumes:
       - name: secret-volume
         secret:
           secretName: test-db-secret
       containers:
       - name: db-client-container
         image: myClientImage
         volumeMounts:
         - name: secret-volume
           readOnly: true
           mountPath: "/etc/secret-volume"
   EOF
   ```

   Note:

   How the specs for the two Pods differ only in one field; this facilitates creating Pods with different capabilities from a common Pod template.

4. Apply all those objects on the API server by running:

   ```
   kubectl create -f pod.yaml
   ```

Both containers will have the following files present on their filesystems with the values for each container's environment:

```
/etc/secret-volume/username
/etc/secret-volume/password
```

You could further simplify the base Pod specification by using two service accounts:

1. `prod-user` with the `prod-db-secret`
2. `test-user` with the `test-db-secret`

The Pod specification is shortened to:

```
apiVersion: v1
kind: Podmetadata:  name: prod-db-client-pod  labels:    name: prod-db-clientspec:  serviceAccount: prod-db-client  containers:  -
```

References

- Secret
- Volume
- Pod

What's next

- Learn more about Secrets.
- Learn about Volumes.

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:

- iximiuz Labs
- Killercoda
- KodeKloud
- 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`, and the `args` field corresponds to `CMD` 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 

```
apiVersion: v1
kind: Podmetadata:  name: command-demo  labels:    purpose: demonstrate-commandspec:  containers:  - name: command-demo-container
```

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.

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

Define Environment Variable Values Using An Init Container

Expose Pod Information to Containers Through Environment Variables

Expose Pod Information to Containers Through Files

Distribute Credentials Securely Using Secrets

Expose Pod Information to Containers Through Files

This page shows how a Pod can use a `downwardAPI` volume, to expose information about itself to containers running in the Pod. A `downwardAPI` volume can expose Pod fields and container fields.

In Kubernetes, there are two ways to expose Pod and container fields to a running container:

- Environment variables
- Volume files, as explained in this task

Together, these two ways of exposing Pod and container fields are called the *downward API*.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Store Pod fields

In this part of exercise, you create a Pod that has one container, and you project Pod-level fields into the running container as files. Here is the manifest for the Pod:

pods/inject/dapi-volume.yaml  Copy pods/inject/dapi-volume.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: kubernetes-downwardapi-volume-example  labels:    zone: us-est-coast    cluster: test-cluster1    rack: 
```

In the manifest, 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 defines a `downwardAPI` volume. 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 made Pod-level fields accessible using the downward API. In this next exercise, you are going to pass fields that are part of the Pod definition, but taken from the specific container rather than from the Pod overall. Here is a manifest for a Pod that again has just one container:

pods/inject/dapi-volume-resources.yaml Copy pods/inject/dapi-volume-resources.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: kubernetes-downwardapi-volume-example-2spec:  containers:    - name: client-container      image: regist
```

In the manifest, you can see that the Pod has a `downwardAPI` volume, and that the single container in that Pod mounts the volume at `/etc/podinfo`.

Look at the `items` array under `downwardAPI`. Each element of the array defines a file in the downward API volume.

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 published as a file named `cpu_limit`. The `divisor` field is optional and has the default value of `1`. A divisor of 1 means cores for `cpu` resources, or bytes for `memory` resources.

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:

```
# Run this in a shell inside the container
cat /etc/podinfo/cpu_limit
```

You can use similar commands to view the `cpu_request`, `mem_limit` and `mem_request` files.

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.

What's next

- Read the `spec` API definition for Pod. This includes the definition of Container (part of Pod).
- Read the list of available fields that you can expose using the downward API.

Read about volumes in the legacy API reference:

- 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.

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:

- iximiuz Labs
- Killercoda
- KodeKloud
- 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 a dependent environment variable with common usage defined. Here is the configuration manifest for the Pod:

pods/inject/dependent-envars.yaml Copy pods/inject/dependent-envars.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dependent-envars-demospec:  containers:    - name: dependent-envars-demo      args:        - while true;
```

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.

Note that order matters in the `env` list. An environment variable is not considered "defined" if it is specified further down the list. That is why `UNCHANGED_REFERENCE` fails to resolve `$(PROTOCOL)` in the example above.

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.

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, using the *downward API*. You can use environment variables to expose Pod fields, container fields, or both.

In Kubernetes, there are two ways to expose Pod and container fields to a running container:

- *Environment variables*, as explained in this task
- Volume files

Together, these two ways of exposing Pod and container fields are called the downward API.

As Services are the primary mode of communication between containerized applications managed by Kubernetes, it is helpful to be able to discover them at runtime.

Read more about accessing Services here.

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:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Use Pod fields as values for environment variables

In this part of exercise, you create a Pod that has one container, and you project Pod-level fields into the running container as environment variables.

pods/inject/dapi-envars-pod.yaml Copy pods/inject/dapi-envars-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dapi-envars-fieldrefspec:  containers:    - name: test-container      image: registry.k8s.io/busybox:1.2
```

In that manifest, you can see five environment variables. The `env` field is an array of environment variable definitions. 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:

```
# If the new Pod isn't yet healthy, rerun this command a few times.
```

```
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:

```
# Run this in a shell inside the container
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 information from Pod-level fields as the values for environment variables. In this next exercise, you are going to pass fields that are part of the Pod definition, but taken from the specific container rather than from the Pod overall.

Here is a manifest for another Pod that again has just one container:

pods/inject/dapi-envars-container.yaml Copy pods/inject/dapi-envars-container.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: dapi-envars-resourcefieldrefspec:  containers:    - name: test-container      image: registry.k8s.io/busy
```

In this manifest, you can see four environment variables. The `env` field is an array of environment variable definitions. 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 fields that are specific to this container.

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:

```
# If the new Pod isn't yet healthy, rerun this command a few times.
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

- Read Defining Environment Variables for a Container
- Read the `spec` API definition for Pod. This includes the definition of Container (part of Pod).
- Read the list of available fields that you can expose using the downward API.

Read about Pods, containers and environment variables in the legacy API reference:

- PodSpec
- Container
- EnvVar
- EnvVarSource
- ObjectFieldSelector
- ResourceFieldSelector

Define Environment Variable Values Using An Init Container

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

This page show how to configure environment variables for containers in a Pod via 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:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be version v1.34.

To check the version, enter `kubectl version`.

How the design works

In this exercise, you will create a Pod that sources environment variables from files, projecting these values into the running container.

pods/inject/envars-file-container.yaml  Copy pods/inject/envars-file-container.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: envfile-test-podspec:  initContainers:    - name: setup-envfile      image:  nginx      command: ['sh',
```

In this manifest, you can see the `initContainer` mounts an `emptyDir` volume and writes environment variables to a file within it, and the regular containers reference both the file and the environment variable key through the `fileKeyRef` field without needing to mount the volume. When `optional` field is set to false, the specified `key` in `fileKeyRef` must exist in the environment variables file.

The volume will only be mounted to the container that writes to the file (`initContainer`), while the consumer container that consumes the environment variable will not have the volume mounted.

The env file format adheres to the kubernetes env file standard.

During container initialization, the kubelet retrieves environment variables from specified files in the `emptyDir` volume and exposes them to the container.

Note:

All container types (initContainers, regular containers, sidecars containers, and ephemeral containers) support environment variable loading from files.

While these environment variables can store sensitive information, `emptyDir` volumes don't provide the same protection mechanisms as dedicated Secret objects. Therefore, exposing confidential environment variables to containers through this feature is not considered a security best practice.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/envars-file-container.yaml
```

Verify that the container in the Pod is running:

```
# If the new Pod isn't yet healthy, rerun this command a few times.
kubectl get pods
```

Check container logs for environment variables:

```
kubectl logs dapi-test-pod -c use-envfile | grep DB_ADDRESS
```

The output shows the values of selected environment variables:

```
DB_ADDRESS=address
```

Env File Syntax

The format of Kubernetes env files originates from `.env` files.

In a shell environment, `.env` files are typically loaded using the `source .env` command.

For Kubernetes, the defined env file format adheres to stricter syntax rules:

- Blank Lines: Blank lines are ignored.
- Leading Spaces: Leading spaces on all lines are ignored.
- Variable Declaration: Variables must be declared as `VAR=VAL`. Spaces surrounding = and trailing spaces are ignored.

  `VAR=VAL → VAL`

- Comments: Lines beginning with # are treated as comments and ignored.

  ```
  # comment
  VAR=VAL → VAL

  VAR=VAL # not a comment → VAL # not a comment
  ```

- Line Continuation: A backslash (\) at the end of a variable declaration line indicates the value continues on the next line. The lines are joined with a single space.

  ```
  VAR=VAL \
  VAL2
  → VAL VAL2
  ```

What's next

- Learn more about environment variables.
- Read Defining Environment Variables for a Container
- Read Expose Pod Information to Containers Through Environment Variables

---

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:

- iximiuz Labs
- Killercoda
- KodeKloud
- 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.

The `env` and `envFrom` fields have different effects.

`env`
: allows you to set environment variables for a container, specifying a value directly for each variable that you name.

`envFrom`
: allows you to set environment variables for a container by referencing either a ConfigMap or a Secret. When you use `envFrom`, all the key-value pairs in the referenced ConfigMap or Secret are set as environment variables for the container. You can also specify a common prefix string.

You can read more about ConfigMap and Secret.

This page explains how to use `env`.

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 Copy pods/inject/envars.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: envar-demo  labels:    purpose: demonstrate-envarsspec:  containers:  - name: envar-demo-container    im
```

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 under `.spec.containers[*].env[*]` 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. The environment variable `MESSAGE` combines the set of all these environment variables and then uses it as a CLI argument passed to the `env-print-demo` container.

Environment variable names may consist of any printable ASCII characters except '='.

```
apiVersion: v1
kind: Podmetadata:  name: print-greetingspec:  containers:  - name: env-print-demo    image: bash    env:    - name: GREETING
```

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.