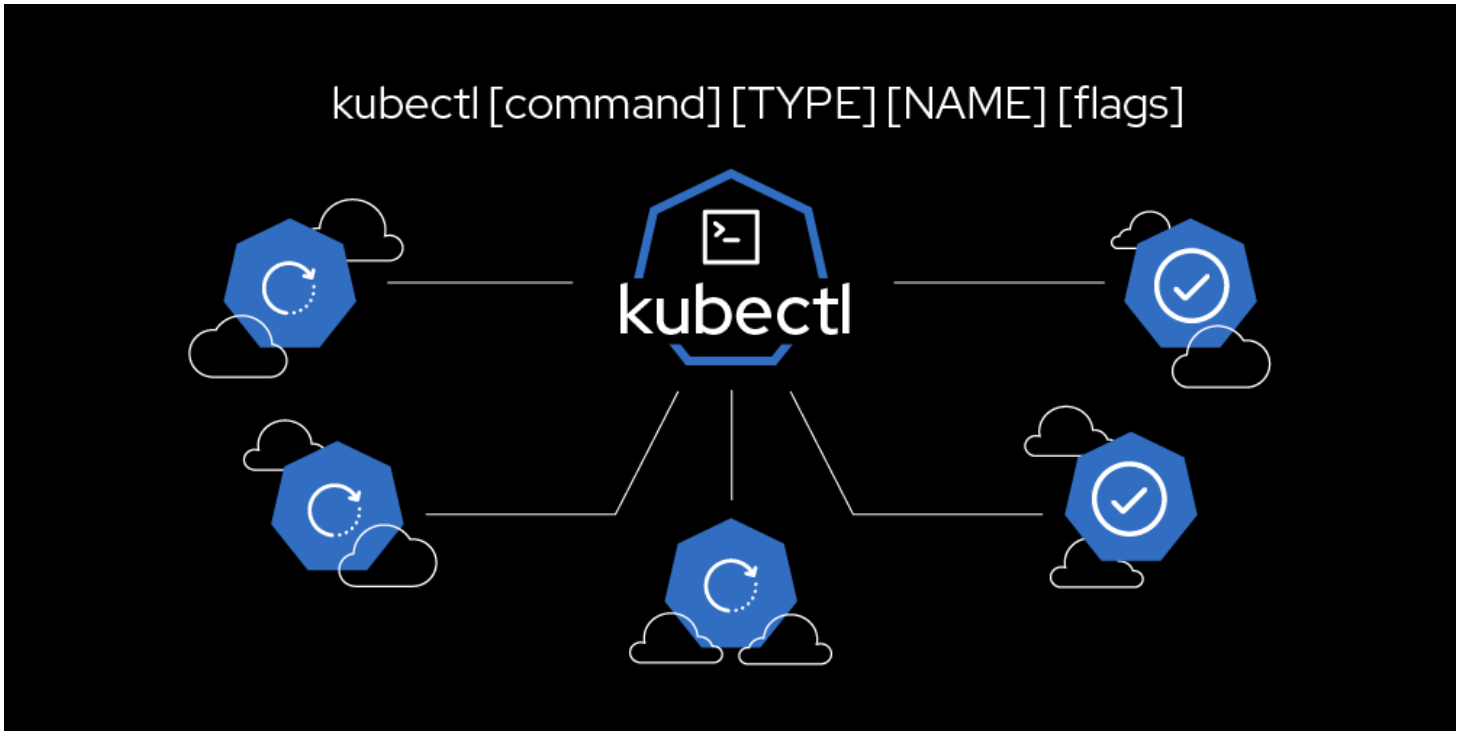
Kubectl: Developer tips for the Kubernetes command line

By Maciej Szulik
November 20, 2020



Kubectl, the Kubernetes command-line interface (CLI), has more capabilities than many developers realize. For example, did you know that `kubectl` can reach the Kubernetes API while running inside a cluster? You can also use `kubectl` to assume different user identities, to select a custom editor to run with the `kubectl edit` command, and more.

In this article, I introduce several `kubectl` CLI features that will improve your daily workflow. You'll also learn about the new `kubectl debug` command in Kubernetes 1.20.

In-cluster configuration

When `kubectl` needs to locate a configuration file, it checks several places. You are probably familiar with the `$HOME/.kube/` directory, which is the default directory where `kubectl` stores its configuration and cache files. You've also heard about the `--kubeconfig` flag, or `KUBECONFIG` environment variable, which is used to pass the location of a configuration file.

Another location that `kubectl` checks when loading files is the in-cluster configuration. Not many users know about this option, so I'll use an example to demonstrate how it works.

Get pods from the container

To start, we'll run a simple `centos:7` container image:

```
$ kubectl run centos --stdin --tty --image=centos:7
```

I passed the `--stdin` and `--tty` flags to attach to the pod as soon as it is running. We also need a `kubectl` binary in the pod:

```
$ kubectl cp kubectl centos:/bin/
```

Now, let's see what happens when we try a `get pods` command on the CentoOS 7 container:

```
$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:soltys
```

This error says that my user does not have the necessary permissions for this operation. If you look closely at the username (`system:serviceaccount:soltysh:default`), you'll notice it's actually a default service account assigned to my pod. But where did that come from? To find out, we can increase the verbosity of `kubectl` and check the debugging information:

```
$ kubectl get pods -v=4
I1104 11:51:21.969428      57 merged_client_builder.go:163] Using in-cluster namespa
```

Notice that I've used the in-cluster configuration for this command. The in-cluster configuration checks for a service account token located in `/var/run/secrets/kubernetes.io/serviceaccount/token` . It also checks the two environment variables `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT` . When it finds all three of these, it knows that it is running inside of a Kubernetes cluster. It then knows that it should read the injected data to talk to the cluster.

Using the view role for read access

Most default Kubernetes clusters with role-based access control (RBAC) turned on have a pre-existing set of cluster roles. Here, we'll use the `view` role, which has read access to all non-escalating resources. To use this role, we need to create a `ClusterRoleBinding`, like this:

```
$ kubectl create clusterrolebinding view-soltysh --clusterrole=view --serviceaccount
```

Now, when we re-run the `get pods` command, we should be able to view all of the pods running in the current namespace:

```
$ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
centos   1/1     Running   0          18m
```

The operation succeeded because we used a role that allowed us to view the pods.

> **Note**: If you are wondering where the namespace defaulting happened, check the contents of the `/var/run/secrets/kubernetes.io/serviceaccount/namespace` file.

User impersonation with the `--as=user` flag

Kubernetes has capabilities similar to the `sudo` command for Unix. This feature, called user impersonation, lets you invoke any command as a different user. To use this feature in `kubectl`, you need to specify the `--as=user` flag, where `user` is the name of the user you wish to impersonate. Once again, an example will demonstrate the concept.

Permission to `impersonate`

To set up the demonstration, let's start as a non-cluster admin user. We will attempt to get the pods from the namespace that we used for the previous example:

```
$ KUBECONFIG=nonadmin kubectl get pods -n soltysh
Error from server (Forbidden): pods is forbidden: User "nonadmin" cannot list resour
```

As expected, we receive an error that we don't have access to that namespace. Now, let's try the same command using the `--as=system:admin` flag. This lets us impersonate `system:admin`:

```
$ KUBECONFIG=nonadmin kubectl get po -n soltysh --as=system:admin
Error from server (Forbidden): users "system:admin" is forbidden: User "nonadmin" ca
```

This error tells us that we don't have the necessary permissions for a user impersonation. Namely, we need to perform the 'impersonate' verb on the user attribute, but we currently lack the permissions to do it. How can we access the permissions that we need?

The `impersonator` cluster role

In my cluster, I have an `impersonator` cluster role. To use the `impersonate` verb, we need to assign this role to the current user. The trick is to invoke the `impersonator` operation as a `cluster-admin`:

```
$ kubectl create clusterrolebinding impersonator-nonadmin --clusterrole=impersonator
```

Now, when we re-run the command as a non-admin user, we can read pods from `soltysh`'s namespace:

```
$ KUBECONFIG=nonadmin kubectl get po -n soltysh --as=system:admin
NAME     READY   STATUS    RESTARTS   AGE
centos   1/1     Running   0          36m
```

To confirm that this trick works, let's see what happens when we try getting the pods without the `--as` flag:

```
$ KUBECONFIG=nonadmin kubectl get po -n soltysh
Error from server (Forbidden): pods is forbidden: User "nonadmin" cannot list resour
```

Once again, this command fails. The impersonation only works if we use the `--as` flag correctly.

> **Note**: The `kubectl` CLI also has access to the `--as-group` flag, which allows you to impersonate a group. Furthermore, you can verify how the command works by using `-v=8` with your `kubectl` command. Doing that lets you view all of the headers sent to the cluster.

Specify a custom editor

By default, the `kubectl edit` command assumes that you are using `vi` (on a Unix-flavored system) or Notepad (in Windows) as your editor. If you prefer a different editor, you can use the `KUBE_EDITOR` environment variable to specify it:

```
$ KUBE_EDITOR="code --wait" kubectl edit po/centos -n soltysh --as=system:admin
```

In this case, I'm using Visual Studio Code (VS Code). Note that I've specified the `--wait` flag to ensure that VS Code retains control until all edit operations are complete.

Debug a running application

The last command we'll look at is `alpha`, from Kubernetes 1.19. You can use this command to debug a running application.

The `kubectl alpha debug` command was developed for ephemeral containers but has since transitioned into a full-fledged debugging tool. You can use `kubectl alpha debug` to create any of the following:

- Ephemeral containers for debugging your application (assuming the feature is enabled).
- Copies of your running pods with additional tooling to provide insight in case of application failure.
- Pods that you can use to debug your nodes.

The `kubectl alpha debug` command has many more features for you to check out. Additionally, Kubernetes 1.20 promotes this command to beta. If you use the `kubectl` CLI with Kubernetes 1.20, you'll find the `alpha` command under `kubectl debug`.

Conclusion

I hope the tips I shared in this article are helpful in your daily work. I will leave you with four takeaways:

1. The `kubectl` command knows how to consume in-cluster configurations to communicate with the cluster that it's running in. You need to ensure that you have appropriate access rights for the service account assigned to your pod.
2. The `kubectl --as` flag acts like `sudo` does for Unix-based systems. You need to have the appropriate access rights for the impersonate verb.
3. `KUBE_EDITOR` allows you to choose a different editor for the `kubectl edit` command.
4. Try the new `kubectl debug` command for debugging your applications in Kubernetes 1.20 and higher.

If you have questions about these tools or suggestions for improving them, please reach out to me or other SIG-CLI team members. We are the Kubernetes special interest group for the `kubectl` CLI.