**Configuration Best Practices**

<https://kubernetes.io/docs/concepts/configuration/overview/>

This document highlights and consolidates configuration best practices that are introduced throughout the user guide, Getting Started documentation, and examples.

This is a living document. If you think of something that is not on this list but might be useful to others, please don't hesitate to file an issue or submit a PR.

General Configuration Tips

* When defining configurations, specify the latest stable API version.
* Configuration files should be stored in version control before being pushed to the cluster. This allows you to quickly roll back a configuration change if necessary. It also aids cluster re-creation and restoration.
* Write your configuration files using YAML rather than JSON. Though these formats can be used interchangeably in almost all scenarios, YAML tends to be more user-friendly.
* Group related objects into a single file whenever it makes sense. One file is often easier to manage than several. See the [guestbook-all-in-one.yaml](https://github.com/kubernetes/examples/tree/master/guestbook/all-in-one/guestbook-all-in-one.yaml) file as an example of this syntax.
* Note also that many kubectl commands can be called on a directory. For example, you can call kubectl apply on a directory of config files.
* Don't specify default values unnecessarily: simple, minimal configuration will make errors less likely.
* Put object descriptions in annotations, to allow better introspection.

"Naked" Pods versus ReplicaSets, Deployments, and Jobs

* Don't use naked Pods (that is, Pods not bound to a [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) or [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)) if you can avoid it. Naked Pods will not be rescheduled in the event of a node failure.

A Deployment, which both creates a ReplicaSet to ensure that the desired number of Pods is always available, and specifies a strategy to replace Pods (such as [RollingUpdate](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/" \l "rolling-update-deployment)), is almost always preferable to creating Pods directly, except for some explicit [restartPolicy: Never](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/" \l "restart-policy) scenarios. A [Job](https://kubernetes.io/docs/concepts/workloads/controllers/job/) may also be appropriate.

Services

* Create a [Service](https://kubernetes.io/docs/concepts/services-networking/service/) before its corresponding backend workloads (Deployments or ReplicaSets), and before any workloads that need to access it. When Kubernetes starts a container, it provides environment variables pointing to all the Services which were running when the container was started. For example, if a Service named foo exists, all containers will get the following variables in their initial environment:
* FOO\_SERVICE\_HOST=<the host the Service is running on>
* FOO\_SERVICE\_PORT=<the port the Service is running on>

*This does imply an ordering requirement* - any Service that a Pod wants to access must be created before the Pod itself, or else the environment variables will not be populated. DNS does not have this restriction.

* An optional (though strongly recommended) [cluster add-on](https://kubernetes.io/docs/concepts/cluster-administration/addons/) is a DNS server. The DNS server watches the Kubernetes API for new Services and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all Pods should be able to do name resolution of Services automatically.
* Don't specify a hostPort for a Pod unless it is absolutely necessary. When you bind a Pod to a hostPort, it limits the number of places the Pod can be scheduled, because each <hostIP, hostPort, protocol> combination must be unique. If you don't specify the hostIP and protocol explicitly, Kubernetes will use 0.0.0.0 as the default hostIP and TCP as the default protocol.

If you only need access to the port for debugging purposes, you can use the [apiserver proxy](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/" \l "manually-constructing-apiserver-proxy-urls) or [kubectl port-forward](https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/).

If you explicitly need to expose a Pod's port on the node, consider using a [NodePort](https://kubernetes.io/docs/concepts/services-networking/service/" \l "type-nodeport) Service before resorting to hostPort.

* Avoid using hostNetwork, for the same reasons as hostPort.
* Use [headless Services](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) (which have a ClusterIP of None) for service discovery when you don't need kube-proxy load balancing.

Using Labels

* Define and use [labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) that identify **semantic attributes** of your application or Deployment, such as { app.kubernetes.io/name: MyApp, tier: frontend, phase: test, deployment: v3 }. You can use these labels to select the appropriate Pods for other resources; for example, a Service that selects all tier: frontend Pods, or all phase: test components of app.kubernetes.io/name: MyApp. See the [guestbook](https://github.com/kubernetes/examples/tree/master/guestbook/) app for examples of this approach.

A Service can be made to span multiple Deployments by omitting release-specific labels from its selector. When you need to update a running service without downtime, use a [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/).

A desired state of an object is described by a Deployment, and if changes to that spec are *applied*, the deployment controller changes the actual state to the desired state at a controlled rate.

* Use the [Kubernetes common labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/) for common use cases. These standardized labels enrich the metadata in a way that allows tools, including kubectl and [dashboard](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard), to work in an interoperable way.
* You can manipulate labels for debugging. Because Kubernetes controllers (such as ReplicaSet) and Services match to Pods using selector labels, removing the relevant labels from a Pod will stop it from being considered by a controller or from being served traffic by a Service. If you remove the labels of an existing Pod, its controller will create a new Pod to take its place. This is a useful way to debug a previously "live" Pod in a "quarantine" environment. To interactively remove or add labels, use [kubectl label](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "label).

Using kubectl

* Use kubectl apply -f <directory>. This looks for Kubernetes configuration in all .yaml, .yml, and .json files in <directory> and passes it to apply.
* Use label selectors for get and delete operations instead of specific object names. See the sections on [label selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) and [using labels effectively](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#using-labels-effectively).
* Use kubectl create deployment and kubectl expose to quickly create single-container Deployments and Services. See [Use a Service to Access an Application in a Cluster](https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/) for an example.