Init Containers

This page provides an overview of init containers[1)]: specialized containers that run before app containers in a Pod. Init containers can contain utilities or setup scripts not present in an app image.

You can specify init containers (not to confuse with sidecar containers) in the Pod specification alongside regular `containers`

Understanding init containers

A Pod can have multiple containers running apps within it, but it can also have one or more init containers, which run before the app containers are started.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of `container` items (similar to the app `containers` field and its contents).

The status of the init containers is returned in `.status.initContainerStatuses` field as an array of the container statuses (similar to the `.status.containerStatuses` field).

Differences from regular containers

Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an init container are handled differently.

Regular init containers (excluding sidecar containers) do not support the `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` fields (More on that in a upcoming learning unit). Init containers must run to completion before the Pod can be ready. Sidecar containers continue running during a Pod's lifetime, and *do* support some probes.

If you specify multiple init containers for a Pod, kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual.

Differences from sidecar containers

Init containers run and complete their tasks before the main application container starts. Unlike side containers, init containers are not continuously running alongside the main containers.

Init containers run to completion sequentially, and the main container does not start until all the init containers have successfully completed.

init containers do not support `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` whereas sidecar containers support all these probes[2] to control their lifecycle.

Init containers share the same resources (CPU, memory, network) with the main application containers but do not interact directly with them. They can, however, use shared volumes for data exchange.

Using init containers

Because init containers have separate images from app containers, they have some advantages for start-up related code:

- Init containers can contain utilities or custom code for setup that are not present in an app image. For example, there is no need to make an image FROM another image just to use a tool like `sed`, `awk`, `python`, or `dig` during setup.
- The application image builder and deployer roles can work independently without the need to jointly build a single app image.
- Init containers can run with a different view of the filesystem than app containers in the same Pod. Consequently, they can be given access to Secrets that app containers cannot access.
- Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.
- Init containers can securely run utilities or custom code that would otherwise make an app container image less secure. By keeping unnecessary tools separate you can limit the attack surface of your app container image.

Examples

Here are some ideas for how to use init containers:

- Wait for a Service[3] to be created, using a shell one-line command like:

```
for i in {1..100}; do sleep 1; if nslookup myservice; then exit 0; fi; done; exit 1
```

- Register this Pod with a remote server from the downward API with a command like:

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d 'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
```

- Wait for some time before starting the app container with a command like

```
sleep 60
```

- Clone a Git repository into a Volume[4)]
- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the `POD_IP` value in a configuration and generate the main app configuration file using Jinja.

Init containers in use

This example defines a simple Pod that has two init containers. The first waits for `myservice`, and the second waits for `mydb`. Once both init containers complete, the Pod runs the app container from its `spec` section.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app.kubernetes.io/name: MyApp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local;
do echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local;
do echo waiting for mydb; sleep 2; done"]
```

You can start this Pod by running:

```
kubectl apply -f myapp.yaml
```

The output is similar to this:

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
```

or for more details:

```
kubectl describe -f myapp.yaml
```

More information

- regular containers
- sidecar containers
- Secrets
- Pod-lifecycle

[1)]

see init containers

[2)]

see probes

[3)]

see Service

[4)]

see Volumes