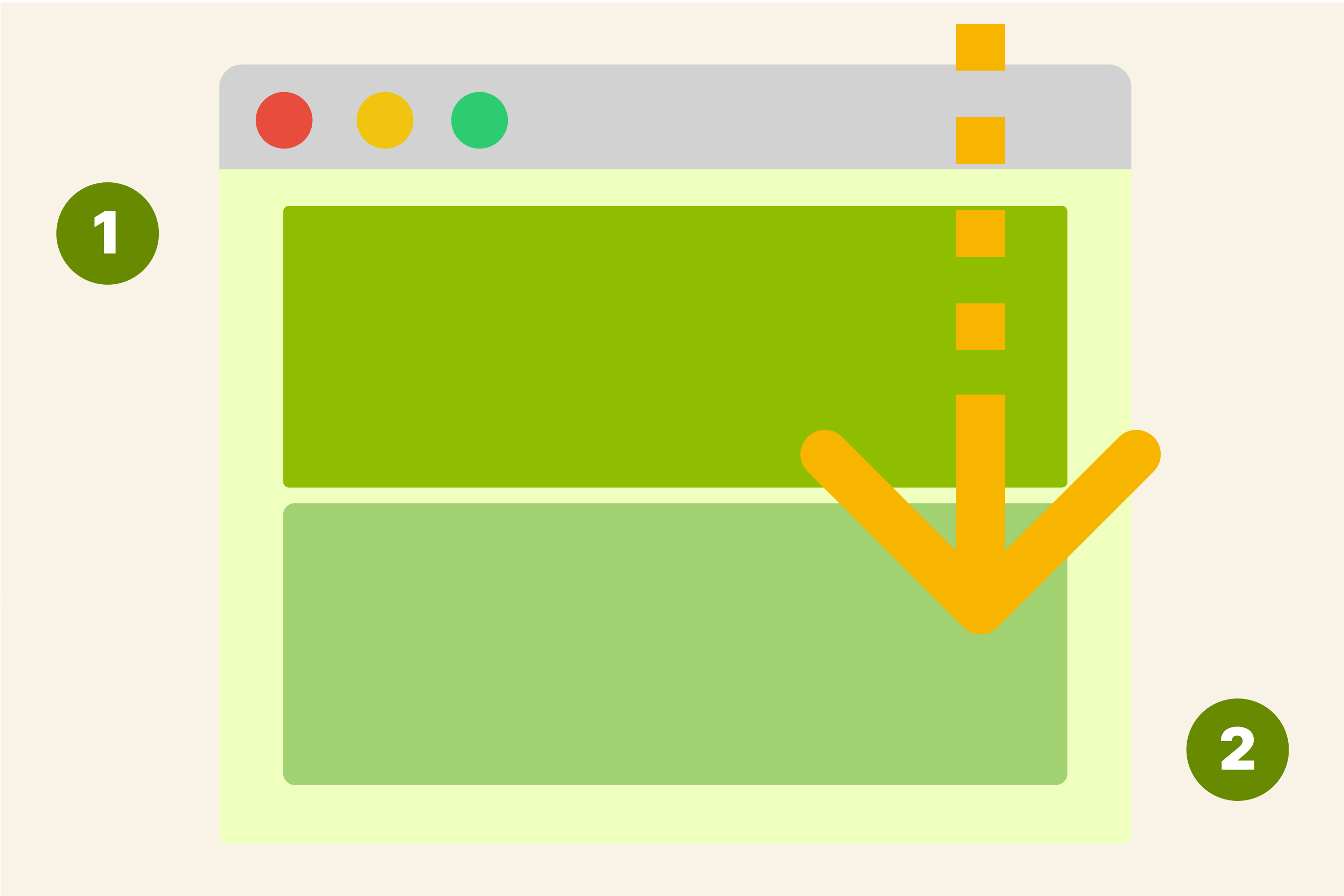
Emanuel Evans

Extending applications on Kubernetes with multi-container pods

PUBLISHED IN FEBRUARY 2021



ll learn how you can use the ambassador, adapter, to extend yours apps in Kubernetes without

ise amount of flexibility and the ability to run a wide

If your applications are cloud-native microservices or 12-factor apps, chances are that running them in Kubernetes will be relatively straightforward.

But what about running applications that weren't explicitly designed to be run in a containerized environment?

Kubernetes can handle these as well, although it may be a bit more work to set up.

One of the most powerful tools that Kubernetes offers to help is the **multi-container pod** (although multi-container pods are also useful for cloud-native apps in a variety of cases, as you'll see).

Why would you want to run multiple containers in a pod?

Multi-container pods allow you to change the behaviour of an application without changing its code.

This can be useful in all sorts of situations, but it's convenient for applications that weren't originally designed to be run in containers.

Let's start with an example.

TTP service

before containers became popular (although it's in Kubernetes nowadays) and can be seen as a ava application designed to run in a virtual

n example application that you'd like to enhance

(not at all production-ready) Elasticsearch es-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elasticsearch
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: elasticsearch
  template:
    metadata:
      labels:
        app.kubernetes.io/name: elasticsearch
    spec:
      containers:
        - name: elasticsearch
          image: elasticsearch:7.9.3
          env:
            - name: discovery.type
              value: single-node
          ports:
            - name: http
              containerPort: 9200
---
apiVersion: v1
kind: Service
```

```
ame: elasticsearch
```

environment variable is necessary to get it

eplica.

Elasticsearch will listen on port 9200 over HTTP by default.

You can confirm that the pod works by running another pod in the cluster and `curl` ing to the `elasticsearch` service:

bash

```
$ kubectl run -it --rm --image=curlimages/curl curl \
  -- curl http://elasticsearch:9200
{
  "name" : "elasticsearch-77d857c8cf-mk2dv",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "z98oL-w-SLKJBhh5KVG4kg",
  "version" : {
    "number" : "7.9.3",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "c4138e51121ef06a6404866cddc601906fe5c868",
    "build_date" : "2020-10-16T10:36:16.141335Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

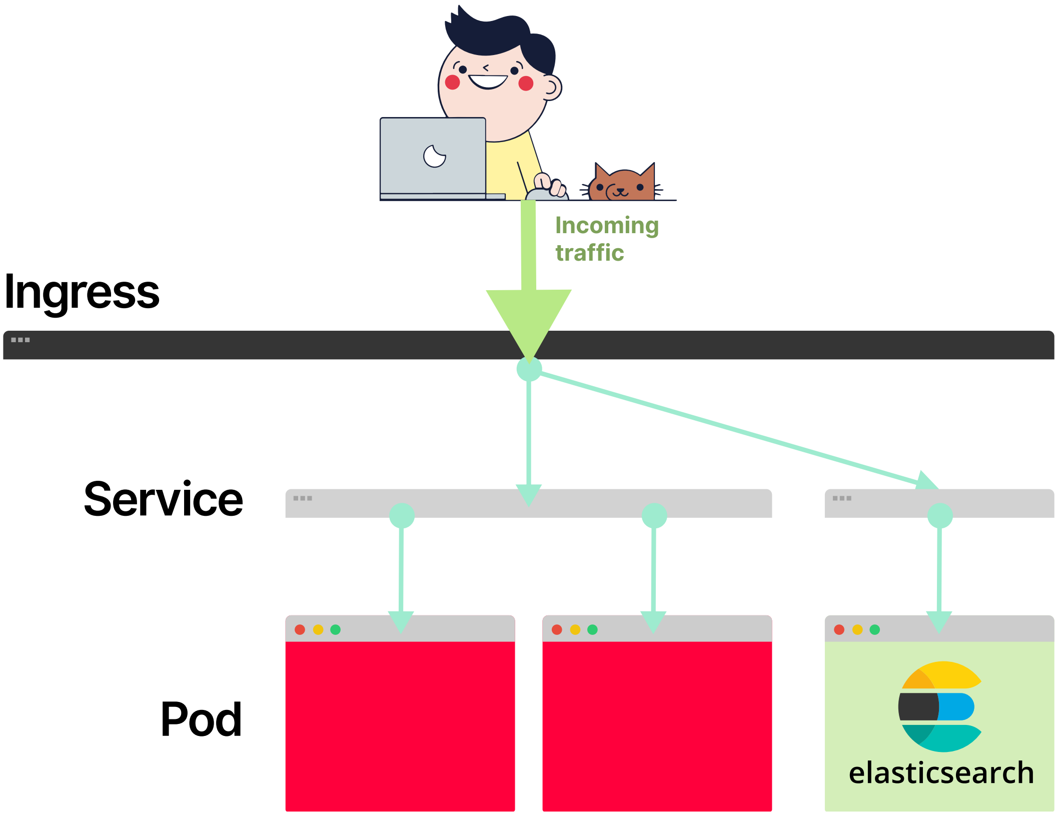

ving towards a [zero-trust security model] and you'd he network.

is if the application doesn't have native TLS

sticsearch support TLS, but it was a paid extra

Our first thought might be to do TLS termination with an [nginx ingress](), since the ingress is the component routing the external traffic in the cluster.

But that won't meet the requirements, since traffic between the ingress pod and the Elasticsearch pod could go over the network unencrypted.



raffic is routed to the Ingress and then to

NEXT

requirements is to tack an nginx proxy container over TLS.

all the the way from the user to the Pod.

1/2

NEXT >

If you include a proxy container in the pod, you can terminate TLS in the Nginx pod.

Here's what the deployment might look like:

es-secure-deployment.yaml

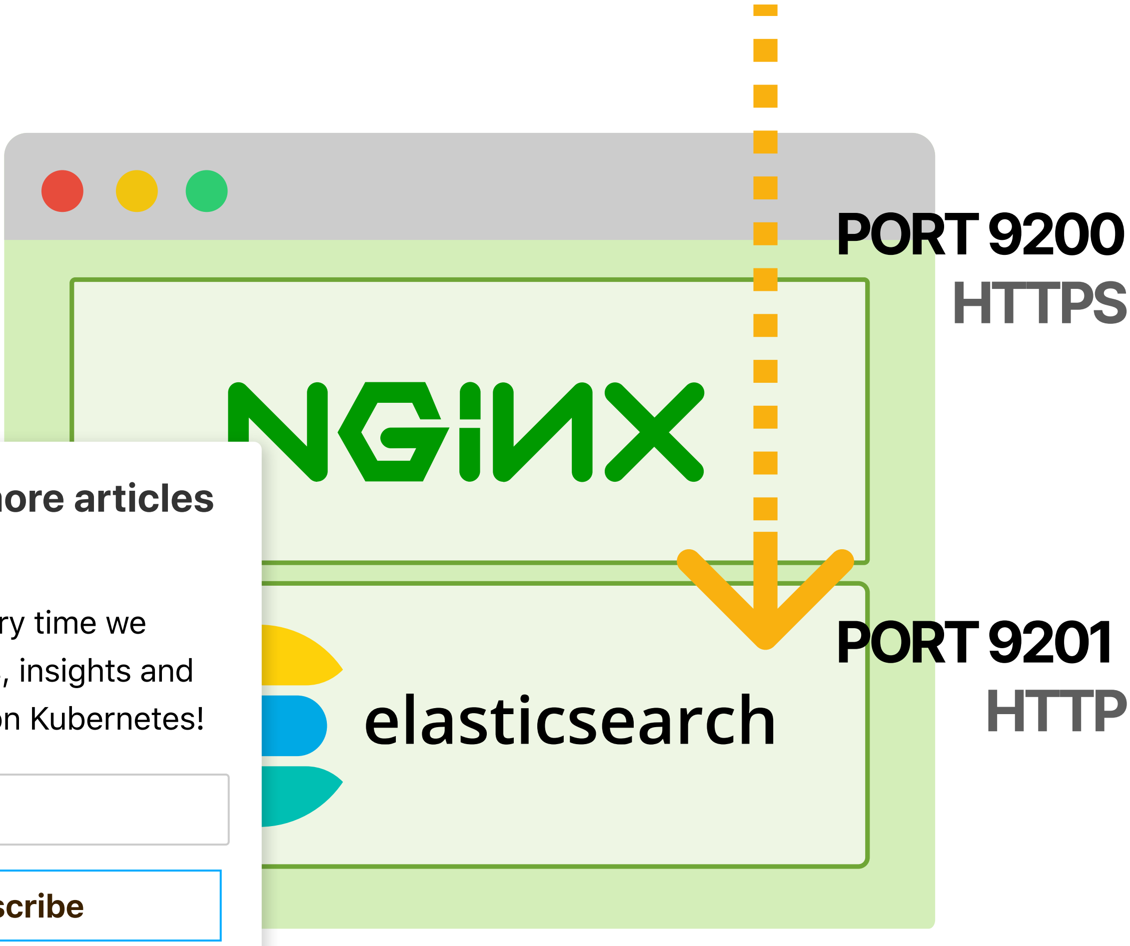

```
/name: elasticsearch
io/name: elasticsearch
search
image: elasticsearch:7.9.3
```

```
          env:
            - name: discovery.type
              value: single-node
            - name: network.host
              value: 127.0.0.1
            - name: http.port
              value: '9201'
        - name: nginx-proxy
          image: nginx:1.19.5
          volumeMounts:
            - name: nginx-config
              mountPath: /etc/nginx/conf.d
              readOnly: true
            - name: certs
              mountPath: /certs
              readOnly: true
          ports:
            - name: https
              containerPort: 9200
      volumes:
        - name: nginx-config
          configMap:
            name: elasticsearch-nginx
        - name: certs
          secret:
            secretName: elasticsearch-tls
---
ginx
;
sticsearch;
 /certs/tls.crt;
_key /certs/tls.key;
http://localhost:9201;
```

Let's unpack that a little bit:

- Elasticsearch is listening on `localhost` on port `9201` instead of the default `0.0.0.0:9200` (that's what the `network.host` and `http.port` environment variables are for).
- The new `nginx-proxy` container listens on port `9200` over HTTPS and proxies requests to Elasticsearch on port `9201`. (The `elasticsearch-tls` secret contains the TLS cert and key, which could be generated with cert-manager, for example.)

So requests from outside the pod will go to Nginx on port 9200 over HTTPS and then forwarded to Elasticsearch on port 9201.



by making an HTTPS request from within the bash

```
$ kubectl run -it --rm --image=curlimages/curl curl \
  -- curl -k https://elasticsearch:9200
{
  "name" : "elasticsearch-5469857795-nddbn",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "XPW9Z8XGTxa7snoUYzeqgg",
  "version" : {
    "number" : "7.9.3",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "c4138e51121ef06a6404866cddc601906fe5c868",
    "build_date" : "2020-10-16T10:36:16.141335Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}

$ _
```

> The `-k` version is necessary for self-signed TLS certificates. In a
> ...t, you'd want to use a trusted certificate.

...ws that the request went through the Nginx proxy:

bash

```
...earch-5469857795-nddbn nginx-proxy | grep curl
.../2020:02:37:07 +0000] "GET / HTTP/1.1" 200 559
```

...'re unable to connect to Elasticsearch over bash

```
$ kubectl run -it --rm --image=curlimages/curl curl \
  -- curl http://elasticsearch:9200
<html>
<head><title>400 The plain HTTP request was sent to HTTPS port</title>
<body>
<center><h1>400 Bad Request</h1></center>
<center>The plain HTTP request was sent to HTTPS port</center>
<hr><center>nginx/1.19.5</center>
</body>
</html>

$ _
```

You've enforced TLS without having to touch the Elasticsearch code or the container image!

Proxy containers are a common pattern

The practice of adding a proxy container to a pod is common enough that it has a name: the **Ambassador Pattern**.

> is post are described in detail in a excellent

only the beginning.

you can do with the Ambassador Pattern:

in the cluster to be encrypted with TLS certificate,
tall an nginx (or other) proxy in every pod in the
o a step farther and use mutual TLS to ensure that
icated as well as encrypted. (This is the primary
meshes such as Istio and Linkerd.)

- You can use a proxy to ensure that a centralized OAuth authority authenticates all requests by verifying jwts. One example of this is gcp-iap-auth, which verifies that requests are authenticated by GCP Identity-Aware Proxy.
- You can connect over a secure tunnel to an external database. This is especially handy for databases that don't have built-in TLS support (like older versions of Redis). Another example is the Google Cloud SQL proxy.

How do multi-container pods work?

Let's take a step back and tease apart the difference between pods and containers on Kubernetes to get a better picture of what's happening under the hood.

A "traditional" container (e.g. one started by `docker run`) provides several forms of isolation:

- Resource isolation (for example, memory limits).
- Process isolation.
- Filesystem and mount isolation.

hings that Docker sets up, but those are the

er the hood are Linux namespaces and control

enient way to limit resources such as CPU or
process can use.

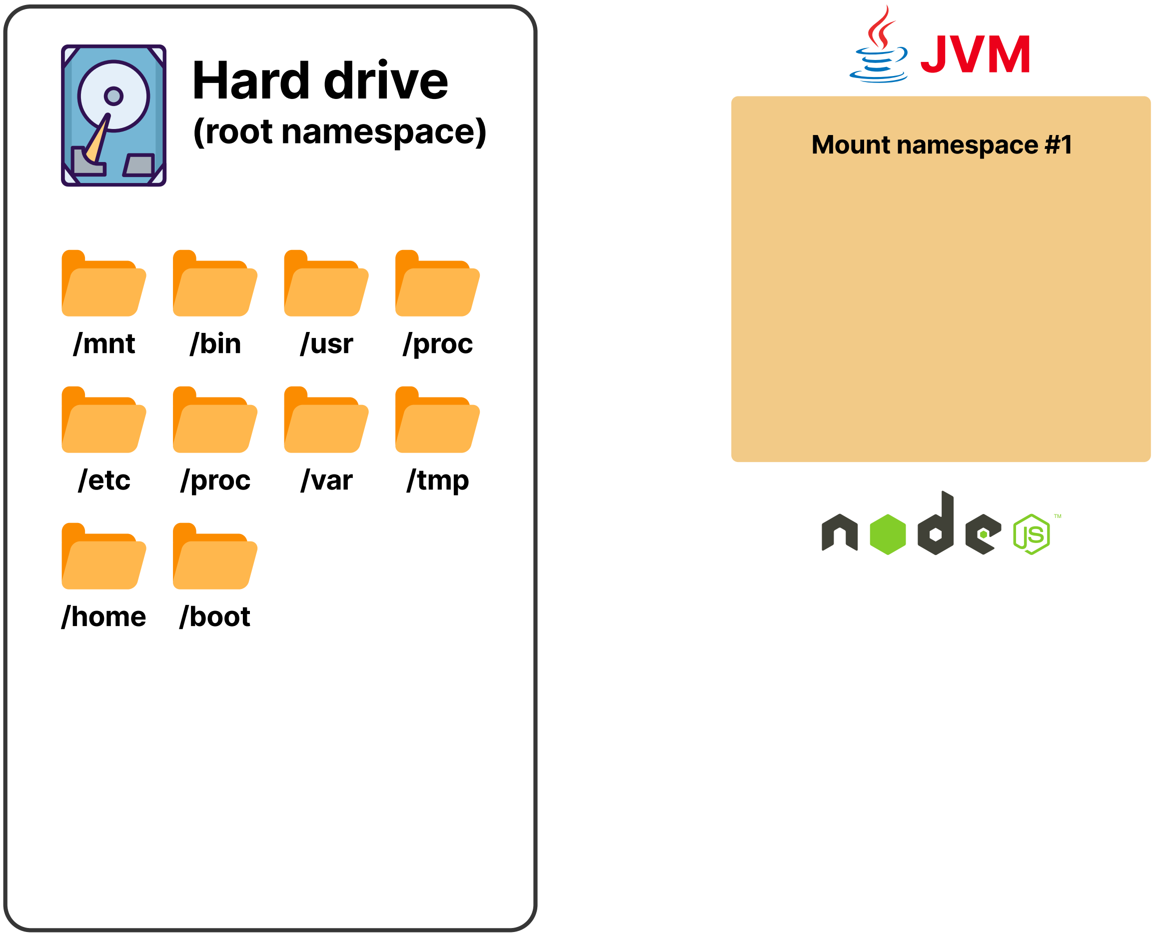
As an example, you could say that your process should use only 2GB of memory and one of your four CPU cores.

Namespaces, on the other hand, are in charge of isolating the process and limiting what it can see.

As an example, the process can only see the network packets that are directly related to it.

It won't be able to see all of the network packets flowing through the network adapter.

Or you could isolate the filesystem and let the process believe that it has access to all of it.



1/4

We're also maintain an active [Telegram, Slack & Twitter community!]

CLOSE

NEXT

...ersion 5.6, there are eight kinds of ...and the mount namespace is one of them.

> If you need a refresher on cgroups and namespaces, [here's an excellent blog post diving into some of the technical details.]

On Kubernetes, a container provides all of those forms of isolation *except* network isolation.

Instead, **network isolation happens at the pod level**.

In other words, each container in a pod will have its filesystem, process table, etc., but all of them will share the same network namespace.

Let's play around with a straightforward multi-pod container to get a better idea of how it works.

pod-multiple-containers.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: podtest
spec:
  containers:
    - name: c1
                        , '5000']

                        hared

                        , '5000']

                        hared
```

Breaking that down a bit:

- There are two containers, both of which just `sleep` for a while.
- There is an [`emptyDir` volume](), which is essentially a temporary local volume that lasts for the lifetime of the pod.
- The `emptyDir` volume is mounted in each pod at the `/shared` directory.

You can see that the volume is mounted on the first container by using `kubectl exec`:

bash

```
$ kubectl exec -it podtest --container c1 -- sh
```

The command attached a terminal session to the container `c1` in the `podtest` pod.

> The `--container` option for `kubectl exec` is often abbreviated `-c`.

You can inspect the volumes attached to `c1` with:

c1@podtest

```
pe ext4 (rw,relatime)
```

mounted on `/shared` — it's the `shared` volume

c1@podtest

```
$ echo "foo" > /tmp/foo
$ echo "bar" > /shared/bar _
```

Let's check the same files from the second container.

First connect to it with:

```bash
$ kubectl exec -it podtest --container c2 -- sh _
```

```
c2@podtest

$ cat /shared/bar
bar
$ cat /tmp/foo
cat: can't open '/tmp/foo': No such file or directory
$ _
```

As you can see, the file created in the `shared` directory is available on both containers, but the file in `/tmp` isn't.

olume, the containers' filesystems are entirely

orking and process isolation.

he network is set up is to use the command `ip` x system's network devices.

in the first container:


```bash
est -c c1 -- ip link
ER_UP> mtu 65536 qdisc noqueue qlen 1000
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
178: eth0@if179: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdi
    link/ether 46:4c:58:6c:da:37 brd ff:ff:ff:ff:ff:ff

$ _
```

And now the same command in the other:

```bash
$ kubectl exec -it podtest -c c2 -- ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
178: eth0@if179: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdi
    link/ether 46:4c:58:6c:da:37 brd ff:ff:ff:ff:ff:ff

$ _
```

You can see that both containers have:

- The same device `eth0` .
- The same MAC addresses `46:4c:58:6c:da:37` .

Since MAC addresses are supposed to be globally unique, this is a clear
re the same device.

ng in action!

ntainer with:

```bash
est -c c1 -- sh _
```

listener with `nc` :

```
c1@podtest
```

```
$ nc -lk -p 5000 127.0.0.1 -e 'date' _
```

The command starts a listener on localhost on port 5000 and prints the `date` command to any connected TCP client.

Can the second container connect to it?

Open a terminal in the second container with:

bash

```
$ kubectl exec -it podtest -c c2 -- sh _
```

Now you can verify that the second container *can* connect to the network listener, but *cannot* see the `nc` process:

c2@podtest

```
$ telnet localhost 5000
Connected to localhost
Sun Nov 29 00:57:37 UTC 2020
Connection closed by foreign host
```

```
MMAND
eep 5000

aux
```

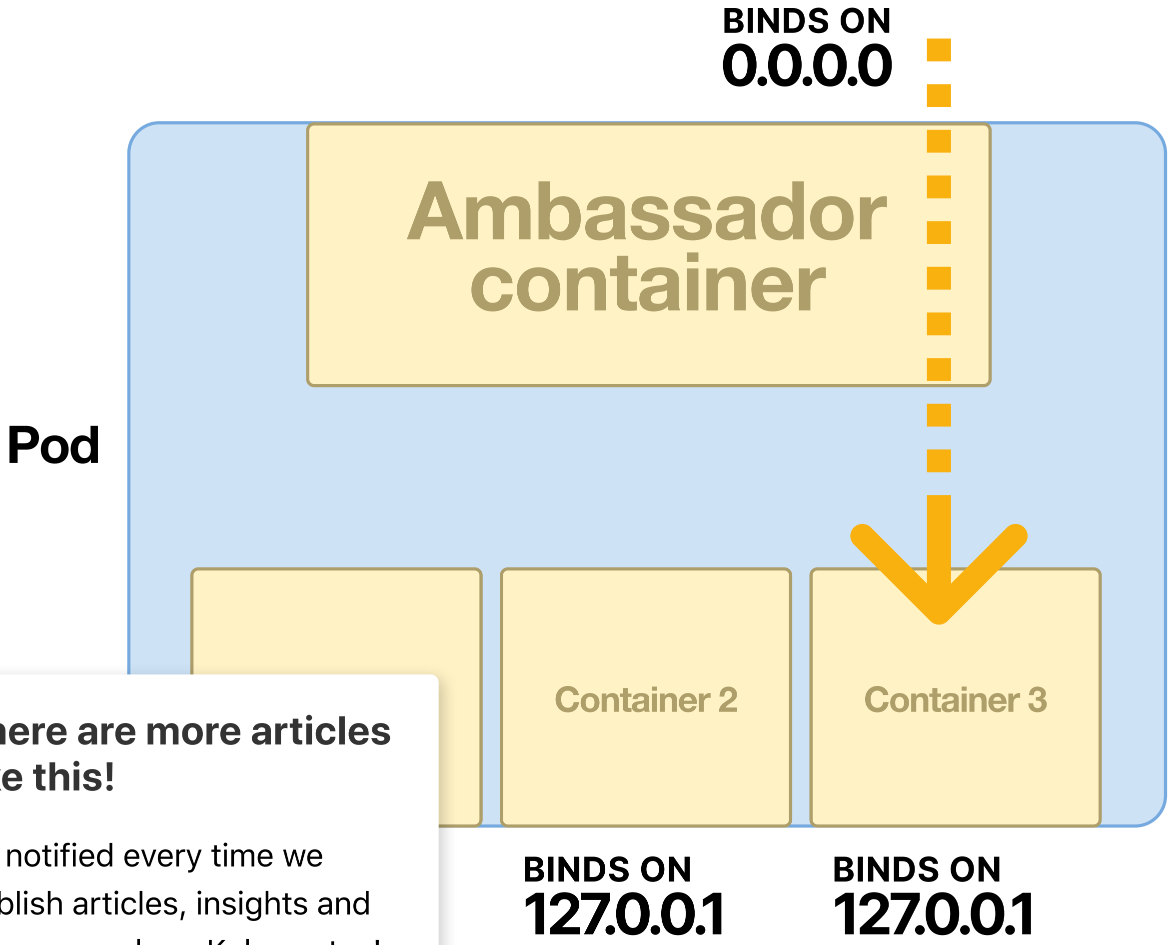
ou can see the output of `date`, which proves that
but `ps aux` (which shows all processes on the
at all.

within a pod have process isolation but not network

This explains how the Ambassador Pattern works:

1. Since all containers share the same network namespace, a single container can listen to all connections — even external ones.
2. The rest of the containers only accept connections from localhost — rejecting any external connection.

The container that receives external traffic is the *Ambassador*, hence the name of the pattern.



…member, though: because the network … multiple containers in a pod can't listen on the …

…her use cases for multi-container pods.

Exposing metrics with a standard interface

Let's say you've standardized on using Prometheus for monitoring all of the services in your Kubernetes cluster, but you're using some applications that don't natively export Prometheus metrics *(for example, Elasticsearch)*.

Can you add Prometheus metrics to your pods without altering your application code?

Indeed you can, using the **Adapter Pattern**.

For the Elasticsearch example, let's add an "exporter" container to the pod that exposes various Elasticsearch metrics in the Prometheus format.

This will be easy, because there's an open-source exporter for Elasticsearch (you'll also need to add the relevant port to the Service):

es-prometheus.yaml


```
/name: elasticsearch

io/name: elasticsearch

search
csearch:7.9.3
```

```yaml
          env:
            - name: discovery.type
              value: single-node
          ports:
            - name: http
              containerPort: 9200
        - name: prometheus-exporter
          image: justwatch/elasticsearch_exporter:1.1.0
          args:
            - '--es.uri=http://localhost:9200'
          ports:
            - name: http-prometheus
              containerPort: 9114
---
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
spec:
  selector:
    app.kubernetes.io/name: elasticsearch
  ports:
    - name: http
      port: 9200
      targetPort: http
    - name: http-prometheus
      port: 9114
                 prometheus
```


you can find the metrics exposed on port 9114:

bash

```
--image=curlimages/curl curl \
rch:9114/metrics | head
eakers_estimated_size_bytes Estimated size in b
eakers_estimated_size_bytes gauge
estimated_size_bytes{breaker="accounting",name=
estimated_size_bytes{breaker="fielddata",name="
estimated_size_bytes{breaker="in_flight_request
_estimated_size_bytes{breaker="model_inference",
```

```
elasticsearch_breakers_estimated_size_bytes{breaker="parent",name="elc
elasticsearch_breakers_estimated_size_bytes{breaker="request",name="el
# HELP elasticsearch_breakers_limit_size_bytes Limit size in bytes for
# TYPE elasticsearch_breakers_limit_size_bytes gauge

$ _
```

Once again, you've been able to alter your application's behaviour without actually changing your code or your container images.

You've exposed standardized Prometheus metrics that can be consumed by cluster-wide tools (like the Prometheus Operator), and have thus achieved a good separation of concerns between the application and the underlying infrastructure.

Tailing logs

Next, let's take a look at the **Sidecar Pattern**, where you add a container to a pod that enhances an application in some way.

The Sidecar Pattern is pretty general and can apply to all sorts of different use
any containers in a pod past the first referred to as

classic sidecar use cases: a log tailing sidecar.

ent, the best practice is to always log to standard
cted and aggregated in a centralized manner.

ns were designed to log to files, and changing
n-trivial.

means you might not have to!

Let's return to Elasticsearch as an example, which is a bit contrived since the Elasticsearch container logs to standard out by default (and it's non-trivial to get it to log to a file).

Here's what the deployment looks like:

sidecar-example.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elasticsearch
  labels:
    app.kubernetes.io/name: elasticsearch
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: elasticsearch
  template:
    metadata:
      labels:
        app.kubernetes.io/name: elasticsearch
    spec:
      containers:
        - name: elasticsearch
          image: elasticsearch:7.9.3

covery.type
ngle-node
h.logs
ar/log/elasticsearch

s
: /var/log/elasticsearch
ging-config
: /usr/share/elasticsearch/config/log4j2.proper
log4j2.properties
 true

p
Port: 9200
```

```
          image: alpine:3.12
          command:
            - tail
            - -f
            - /logs/docker-cluster_server.json
          volumeMounts:
            - name: logs
              mountPath: /logs
              readOnly: true
      volumes:
        - name: logging-config
          configMap:
            name: elasticsearch-logging
        - name: logs
          emptyDir: {}
```

> The logging configuration file is a separate ConfigMap that's too long to include here.

Both containers share a common volume named `logs` .

The Elasticsearch container writes logs to that volume, while the `logs` container just reads from the appropriate file and outputs it to standard out.

am by specifying the appropriate container with

bash

```
earch-6f88d74475-jxdhl logs | head

-29T23:01:42,849Z",

ode",
er-cluster",
search-6f88d74475-jxdhl",

.9.3], pid[7], OS[Linux/5.4.0-52-generic/amd64]
```

```
}
{
  "type": "server",
  "timestamp": "2020-11-29T23:01:42,855Z",
  "level": "INFO",
  "component": "o.e.n.Node",
  "cluster.name": "docker-cluster",
  "node.name": "elasticsearch-6f88d74475-jxdhl",
  "message": "JVM home [/usr/share/elasticsearch/jdk]"
}
{
  "type": "server",
  "timestamp": "2020-11-29T23:01:42,856Z",
  "level": "INFO",
  "component": "o.e.n.Node",
  "cluster.name": "docker-cluster",
  "node.name": "elasticsearch-6f88d74475-jxdhl",
  "message": "JVM arguments […]"
}

$ _
```

The great thing about using a sidecar is that streaming to standard out isn't the only option.

If you needed to switch to a customized log aggregation service, you could just ...er without altering anything else about your

...lecars

...or sidecars; a logging container is only one

...ses you might encounter in the wild:

... in realtime without requiring pod restarts.

...Hashicorp Vault into your application.

- Adding a local Redis instance to your application for low-latency in-memory caching.

Preparing for a pod to run

All of the examples of multi-container pods this post has gone over so far involve several containers running simultaneously.

Kubernetes also provides the ability to run **Init Containers**, which are containers that run to completion before the "normal" containers start.

This allows you to run an initialization script before your pod starts in earnest.

Why would you want your preparation to run in a separate container, instead of (for instance) adding some initialization to your container's entrypoint script?

Let's look to Elasticsearch for a real-world example.

The Elasticsearch docs recommending setting the `vm.max_map_count` sysctl setting in production-ready deployments.

tainerized environments since there's no
or sysctls and any changes have to happen on

cases where you can't customize the Kubernetes

asticsearch in a privileged container, which would
/ to change system settings on its host node, and
add the sysctls.

ly dangerous from a security perspective!

If the Elasticsearch service were ever compromised, an attacker would have root access to its host node.

You can use an init container to mitigate this risk somewhat:

init-es.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elasticsearch
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: elasticsearch
  template:
    metadata:
      labels:
        app.kubernetes.io/name: elasticsearch
    spec:
      initContainers:
        - name: update-sysctl
          image: alpine:3.12
          command: ['/bin/sh']
          args:
            - -c
```

```
               vm.max_map_count=262144
             xt:
               true

             search
             csearch:7.9.3

             covery.type
             ngle-node

             p
             Port: 9200
```

The pod sets the sysctl in a privileged init container, after which the Elasticsearch container starts as expected.

You're still using a privileged container, which isn't ideal, but at least it's extremely minimal and short-lived, so the attack surface is much lower.

> This is the approach recommended by the Elastic Cloud Operator.

Using a privileged init container to prepare a node for running a pod is a fairly common pattern.

For instance, Istio uses init containers to set up iptables rules every time a pod runs.

Another reason to use an init container is to prepare the pod's filesystem in some way.

One common use case is secrets management.

Another init container use case

If you're using something like HashicCorp Vault for secrets management

ts, you can retrieve secrets in an init container and

ptyDir volume.

this:

init-secrets.yaml

```
ame: myapp
```

```yaml
      matchLabels:
        app.kubernetes.io/name: myapp
    template:
      metadata:
        labels:
          app.kubernetes.io/name: myapp
      spec:
        initContainers:
          - name: get-secret
            image: vault
            volumeMounts:
              - name: secrets
                mountPath: /secrets
            command: ['/bin/sh']
            args:
              - -c
              - |
                vault read secret/my-secret > /secrets/my-secret
        containers:
          - name: myapp
            image: myapp
            volumeMounts:
              - name: secrets
                mountPath: /secrets
        volumes:
          - name: secrets
            emptyDir: {}
```


secret will be available on the filesystem for the

systems like the [Vault Agent Sidecar Injector]
a bit more sophisticated in practice (combining
tainers, and sidecars to hide most of the

ner use cases

s you might want to use an init container:

- You want a database migration script to run before your application (this can generally be accomplished in an entrypoint script, but is sometimes easier to do so with a dedicated container).
- You want to retrieve a large file from S3 or GCS that your application depends on (using an init container for this helps to avoid bloat in your application container).

Summary

This post covered quite a lot of ground, so here's a table of some multi-container patterns and when you might want to use them:

Use Case	Ambassador Pattern	Adapter Pattern	Sidecar Pattern	Init Pattern
Encrypt and/or authenticate incoming requests	✅			
[illegible]	✅			
[illegible]		✅		
[illegible]			✅	

Use Case	Ambassador Pattern	Adapter Pattern	Sidecar Pattern	Init Pattern
Add a local Redis cache to your pod			✅	
Monitor and live-reload ConfigMaps			✅	
Inject secrets from Vault into your application			✅	✅
Change node-level settings with a privileged container				✅
Retrieve files from S3 before your application starts				✅

Be sure to read the official documentation and the original container design
dig deeper into this subject.

next article!

en a new article or Kubernetes

etes@gmail.com

*We'll never share your email address, and you can opt-out at any time.

What is Learnk8s?

In-depth Kubernetes training that is practical and easy to understand.

⎈ Instructor-led workshops ›

Deep dive into containers and Kubernetes with the help of our instructors and become an expert in deploying applications at scale.

⎈ Online courses ›

Learn Kubernetes online with hands-on, self-paced courses. No need to leave the comfort of your home.

g ›

ers and Kubernetes

g path — remotely

Team

Careers

Blog

Newsletter

KEEP IN TOUCH

There are more articles like this!

Be notified every time we publish articles, insights and new research on Kubernetes!

Subscribe

We're also maintain an active [Telegram, Slack & Twitter community!]

CLOSE