



Kubernetes Best Practices

Sandeep Dinesh
Developer Advocate
[@sandeepdinesh](#)
[github.com/thesandlord](#)



Kubernetes is really flexible

But you might 🔫 yourself in the 👟



Building Containers

Don't trust arbitrary base images!

Static Analysis of Containers



<https://github.com/coreos/clair>

<https://github.com/banyanops/collector>

Use small base images

Overhead

Node.js App

Your App → 5MB

Your App's Dependencies → 95MB

Total App Size → 100MB

Docker Base Images:

node:8 → 667MB

node:8-wheezy → 521MB

node:8-slim → 225MB

node:8-alpine → 63.7MB

scratch → ~50MB

Overhead

Node.js App

Your App → 5MB

Your App's Dependencies → 95MB

Total App Size → 100MB

← 6.6x App Size!!

Docker Base Images:

node:8 → 667MB

node:8-wheezy → 521MB

node:8-slim → 225MB

node:8-alpine → 63.7MB

scratch → ~50MB

Overhead

Node.js App

Your App → 5MB

Your App's Dependencies → 95MB

Total App Size → 100MB

Docker Base Images:

node:8 → 667MB

node:8-whisperly → 521MB

node:8-slim → 225MB

node:8-alpine → 63.7MB

scratch → ~50MB

← 6.6x App Size!!

← 13.3x "min" overhead!!

Overhead

Node.js App

Your App → 5MB

Your App's Dependencies → 95MB

Total App Size → 100MB

Docker Base Images:

node:18 → 667MB

node:8-whispery → 521MB

node:8-slim → 225MB

node:alpine → 63.7MB

scratch → ~50MB

← 6.6x App Size!!

← 13.3x "min" overhead!!

Pros:

Builds are faster

Need less storage

Cold starts (image pull) are faster

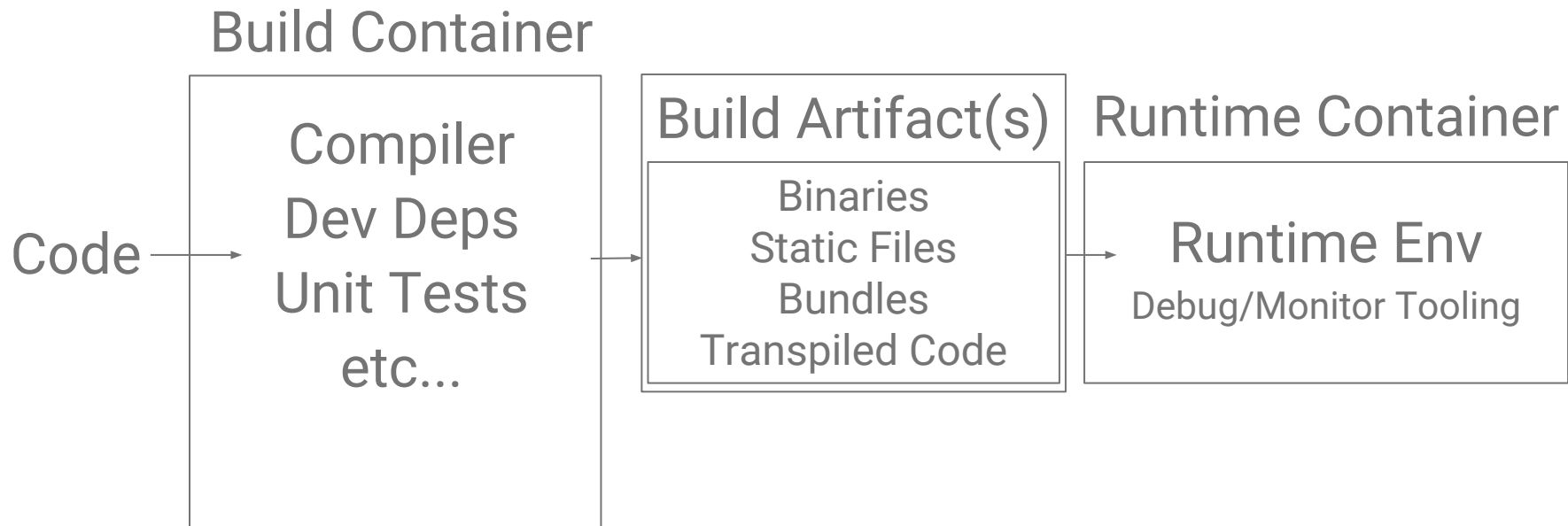
Potentially less attack surface

Cons:

Less tooling inside container

"Non-standard" environment

Use the “builder pattern”



Docker bringing native support for multi-stage builds in Docker CE 17.05



Container Internals

Use a non-root user inside the container

Example Dockerfile

```
FROM node:alpine  
RUN apk update && apk add imagemagick
```

```
RUN groupadd -r nodejs
```

```
RUN useradd -m -r -g nodejs nodejs
```

```
USER nodejs
```

```
ADD package.json package.json
```

```
RUN npm install
```

```
ADD index.js index.js
```

```
CMD npm start
```

Enforce it!

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    # specification of the pod's containers
    # ...
  securityContext:
    runAsNonRoot: true
```

Make the filesystem read-only

Enforce it!

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    # specification of the pod's containers
    # ...
  securityContext:
    runAsNonRoot: true
    readOnlyRootFilesystem: true
```

One process per container

Don't restart on failure. Crash cleanly instead.

Log to stdout and stderr

Add “dumb-init” to prevent zombie processes

Example Dockerfile

```
FROM node:alpine
RUN apk update && apk add imagemagick

RUN groupadd -r nodejs
RUN useradd -m -r -g nodejs nodejs
USER nodejs

ADD https://github.com/Yelp/dumb-init/releases/download/v1.2.0/dumb-init_1.2.0_amd64 \
    /usr/local/bin/dumb-init
RUN chmod +x /usr/local/bin/dumb-init
ENTRYPOINT ["/usr/bin/dumb-init", "--"]

ADD package.json package.json
RUN npm install
ADD index.js index.js
CMD npm start
```

Good News: No need to do this in K8s 1.7



Deployments

Use the “record” option for easier rollbacks

```
$ kubectl apply -f deployment.yaml --record
```

```
...
```

```
$ kubectl rollout history deployments my-deployment
```

```
deployments "ghost-recorded"
```

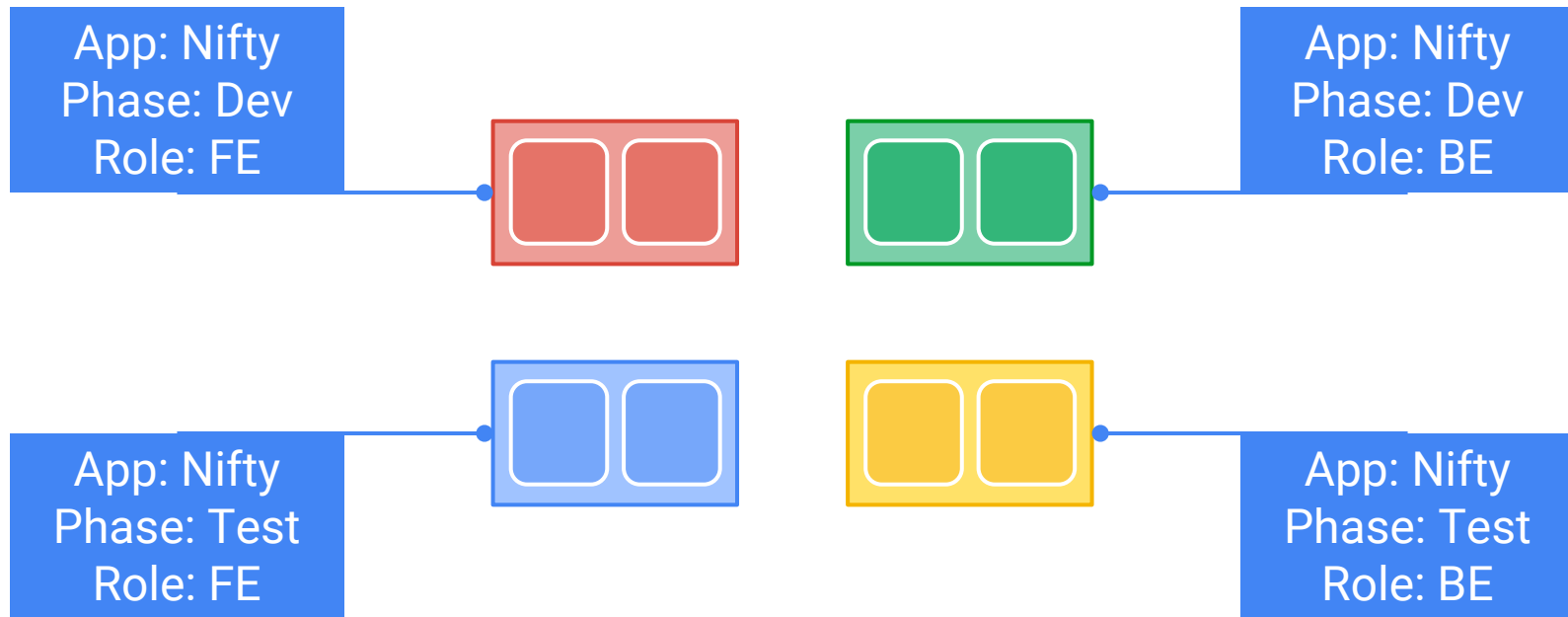
REVISION	CHANGE-CAUSE
1	kubectl apply -f deployment.yaml --record
2	kubectl edit deployments my-deployment
3	kubectl set image deployment/my-deplyoment my-container=app:2.0

Use plenty of descriptive labels

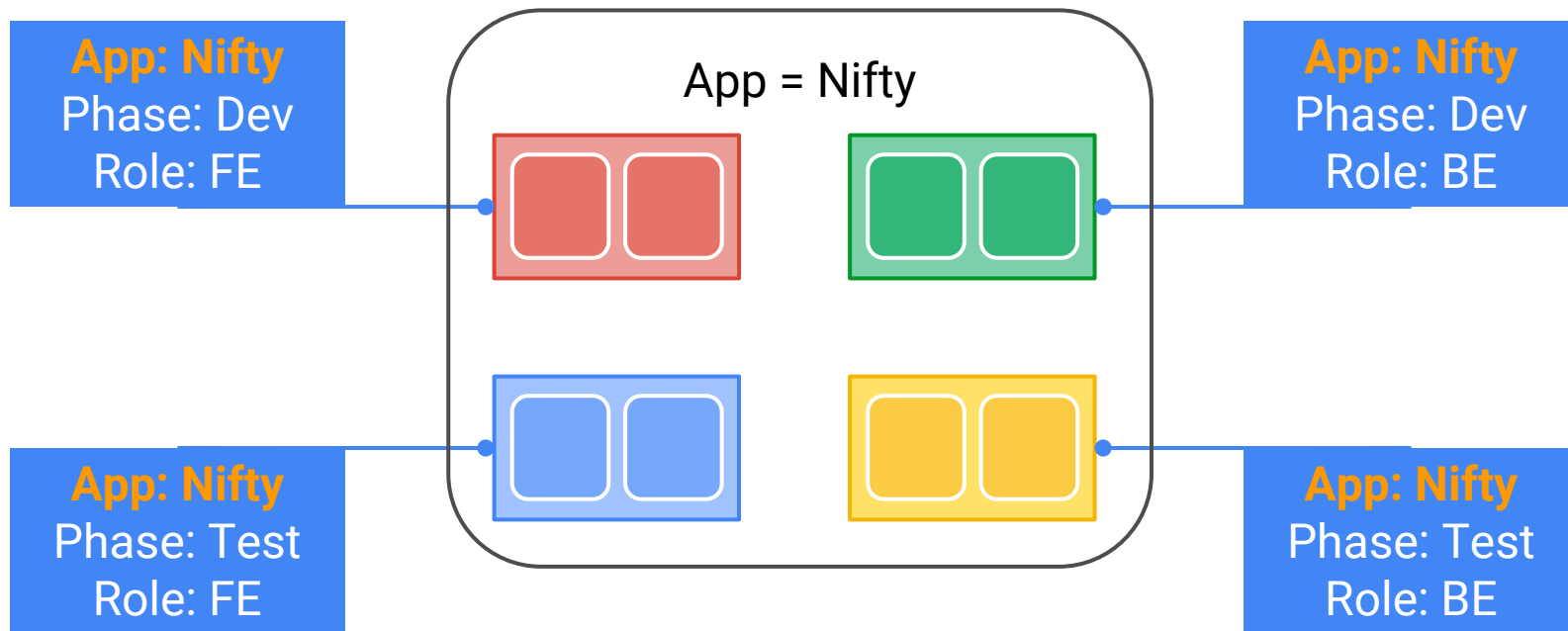
Labels

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 12
  template:
    metadata:
      labels:
        name: web
        color: blue
        experimental: 'true'
```

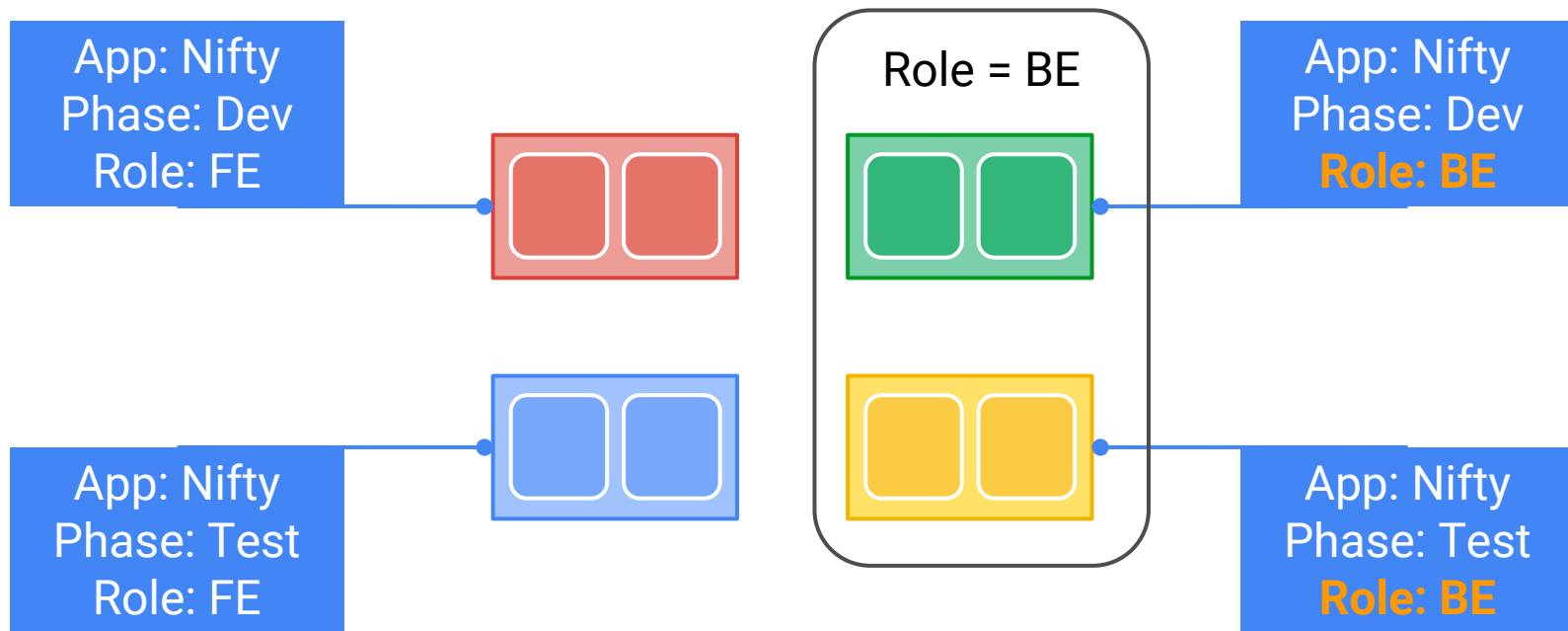
Labels



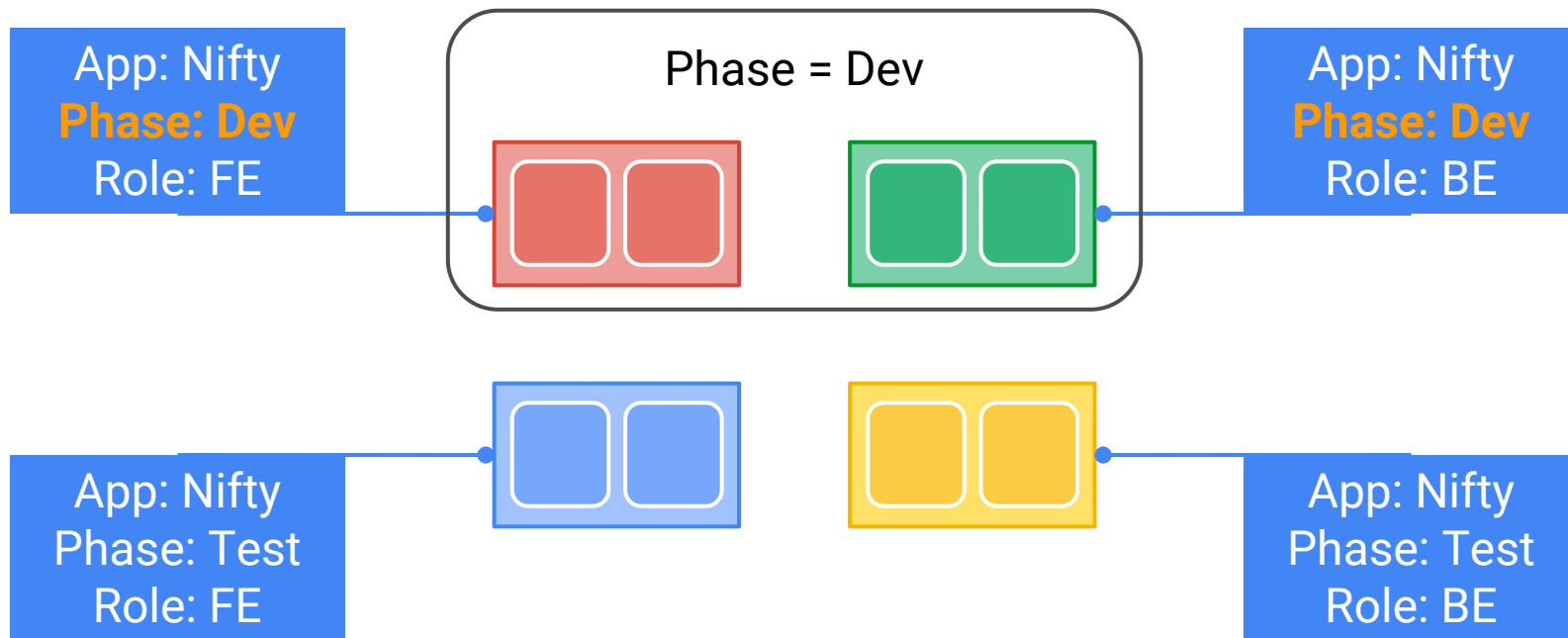
Labels



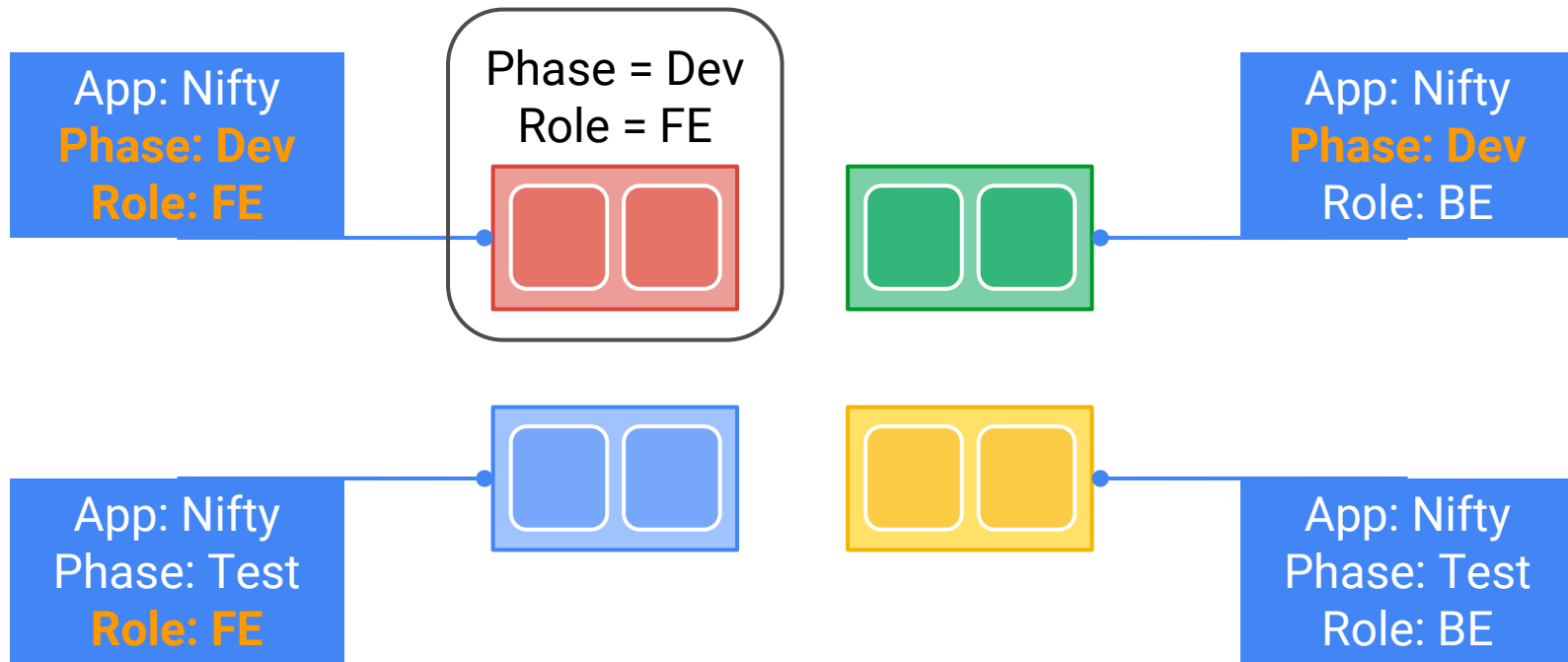
Labels



Labels

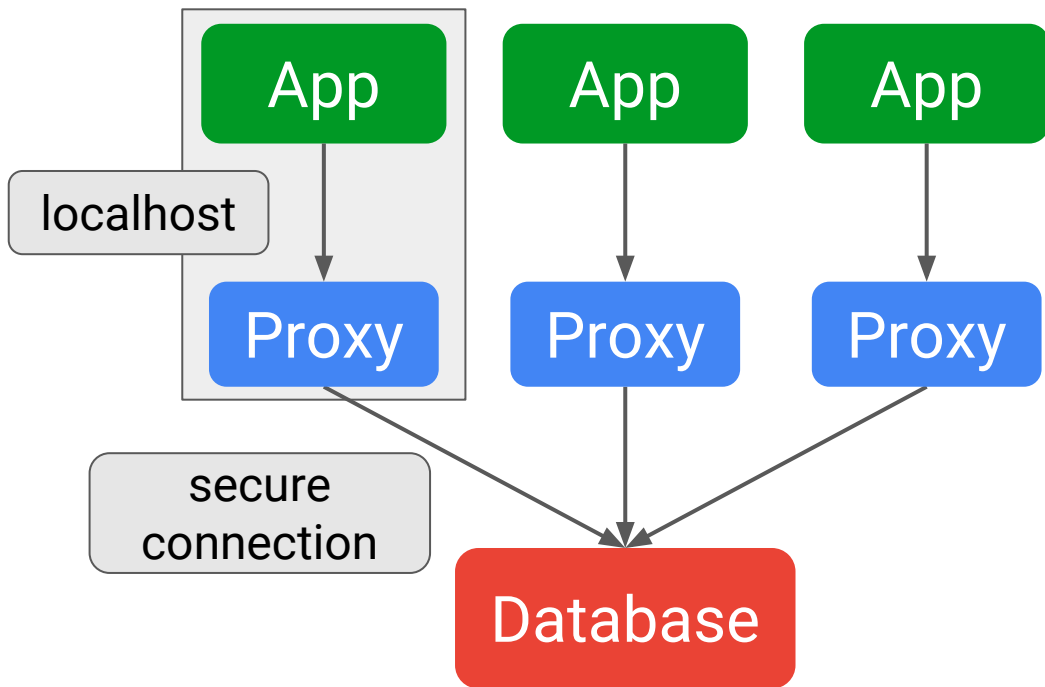


Labels

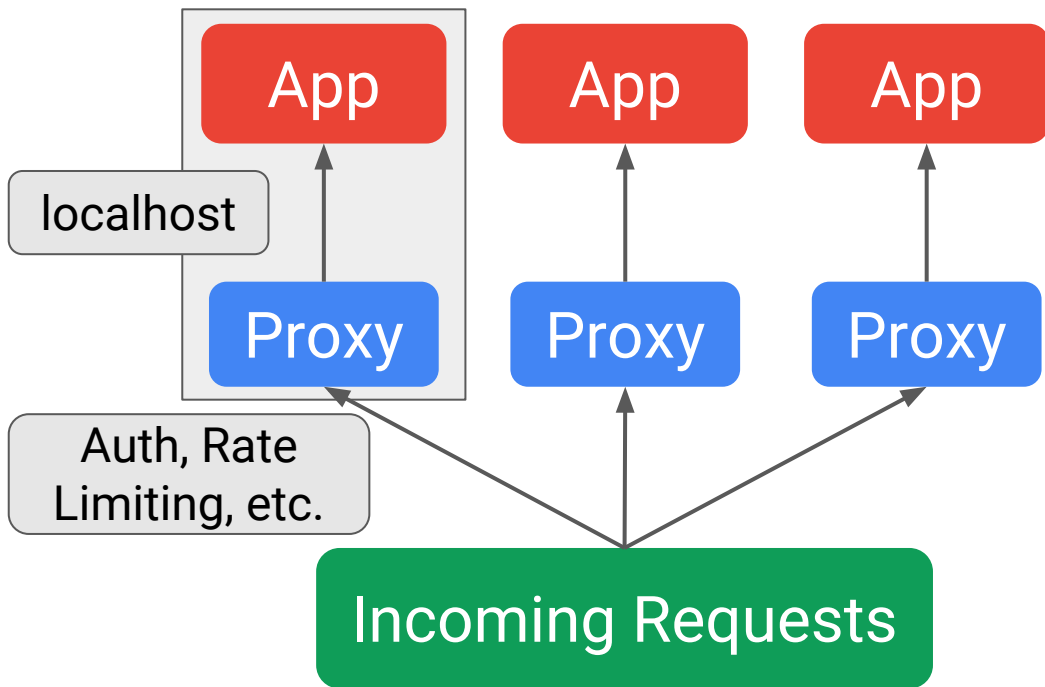


Use sidecar containers for proxies, watchers, etc

Examples



Examples



Don't use sidecars for bootstrapping!

Use init containers instead!

```
apiVersion: v1
kind: Pod
metadata:
  name: awesomeapp-pod
  labels:
    app: awesomeapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
      {
        "name": "init-myapp",
        "image": "busybox",
        "command": ["sh", "-c", "until nslookup myapp; do echo waiting for myapp; sleep 2; done;"]
      },
      {
        "name": "init-mydb",
        "image": "busybox",
        "command": ["sh", "-c", "until nslookup mydb; do echo waiting for mydb; sleep 2; done;"]
      }
    ]'
spec:
  containers:
  - name: awesomeapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600]
```

Don't use `:latest` or no tag

Readiness and Liveness probes are your friend

Health Checks

Readiness → Is the app ready to start serving traffic?

- Won't be added to a service endpoint until it passes
- Required for a “production app” in my opinion

Liveness → Is the app still running?

- Default is “process is running”
- Possible that the process can be running but not working correctly
- Good to define, might not be 100% necessary

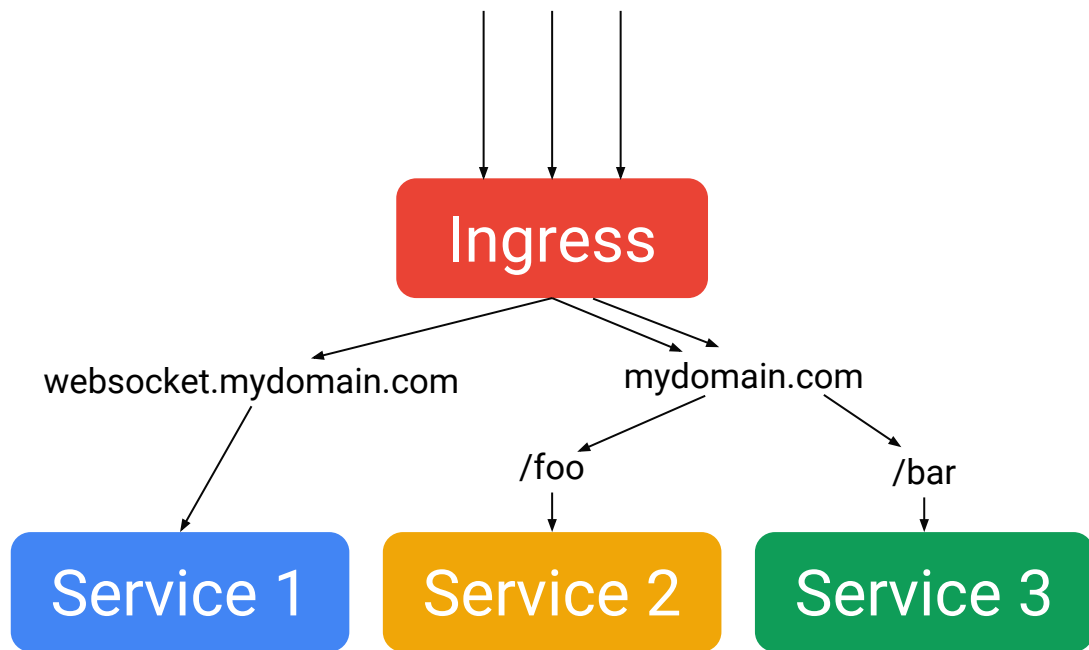
These can sometimes be the same endpoint, but not always



Services

Don't always use `type: LoadBalancer`

Ingress is great



`type: NodePort` can be “good enough”

Use Static IPs. They are free*!

```
$ gcloud compute addresses create ingress --global
...
$ gcloud compute addresses create myservice --region=us-west1
Created ...
address: QQQ.ZZZ.YYY.XXX
...
$
```

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  type: LoadBalancer
  loadBalancerIP: QQQ.ZZZ.YYY.XXX
  ports:
    - port: 80
      targetPort: 3000
      protocol: TCP
  selector:
    name: myapp
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: myingress
  annotations:
    kubernetes.io/ingress.global-static-ip-name: "ingress"
spec:
  backend:
    serviceName: myservice
    servicePort: 80
```

Map external services to internal ones

External Services

Hosted Database

```
kind: Service
apiVersion: v1
metadata:
  name: mydatabase
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
    - port: 12345
```

Database outside cluster but inside network

<pre>kind: Service apiVersion: v1 metadata: name: mydatabase spec: ports: - protocol: TCP port: 80 targetPort: 12345</pre>	<pre>kind: Endpoints apiVersion: v1 metadata: name: mydatabase subsets: - addresses: - ip: 10.128.0.2 ports: - port: 12345</pre>
--	--



Application Architecture

Use Helm Charts

ALL downstream dependencies are unreliable

Make sure your microservices aren't too micro

Use a “Service Mesh”



<https://github.com/istio/istio>



<https://github.com/linkerd/linkerd>

Use a PaaS?





Cluster Management

Use Google Container Engine 🐳

Resources, Anti-Affinity, and Scheduling

Node Affinity

hostname

zone

region

instance-type

os

arch

custom!

Node Taints / Tolerations

special hardware
dedicated hosts
etc

Pod Affinity / Anti-Affinity

hostname

zone

region

Use Namespaces to split up your cluster

Role Based Access Control

Unleash the Chaos Monkey

More Resources

- <http://blog.kubernetes.io/2016/08/security-best-practices-kubernetes-deployment.html>
- <https://github.com/gravitational/workshop/blob/master/k8sprod.md>
- <https://nodesource.com/blog/8-protips-to-start-killing-it-when-dockerizing-node-js/>
- <https://www.ianlewis.org/en/using-kubernetes-health-checks>
- <https://www.linux.com/learn/rolling-updates-and-rollbacks-using-kubernetes-deployments>
- <https://kubernetes.io/docs/api-reference/v1.6/>



Questions?

What best practices do you have?