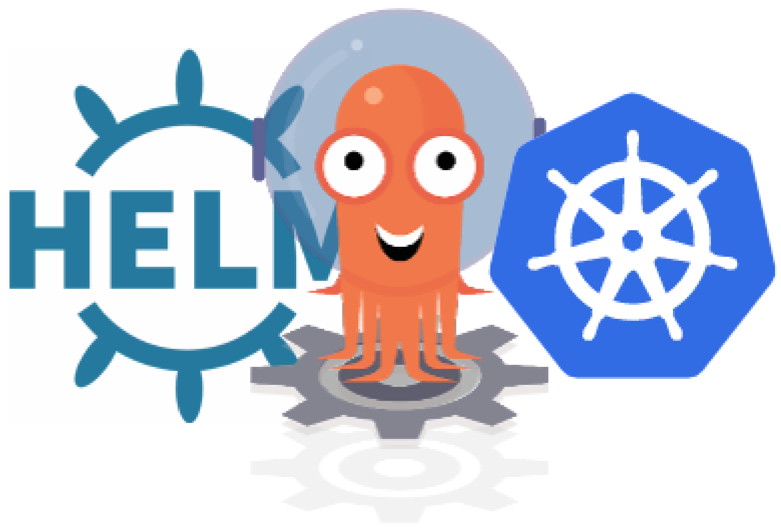
Alex Jones

85 Followers About Follow

Bootstrapping Kubernetes with ArgoCD

Alex Jones Mar 27, 2020 · 6 min read



ArgoCD has quickly become one of the *defacto* tools for GitOps within Kubernetes. I recently learnt about a lesser advertised feature that provides an incredible capability to aid in bootstrapping new Kubernetes cluster from scratch.

When implemented, this allows you to deploy multiple helm-charts and manifest repositories from existing sources upon the spawning of the cluster in a big-bang style setup that can be phased with the power of sync-wave to create a set of pipelines ready to go.

This article is an explanation and demonstration of how you can use ArgoCD to bootstrap Kubernetes services out of the box either locally or on a remote cluster that's been freshly provisioned.

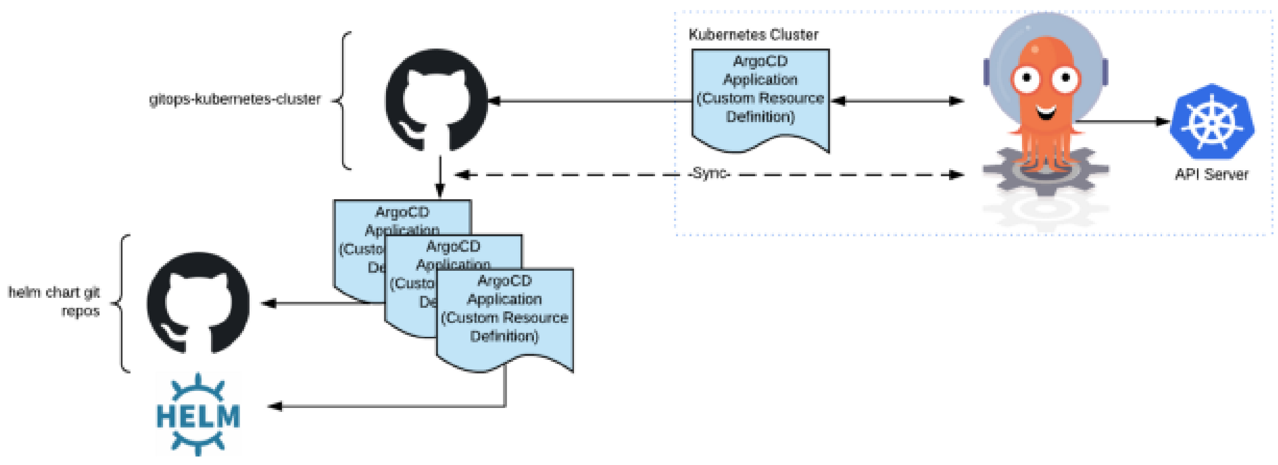
these, there is a demonstration of a particular paradigm of wrapping ArgoCD CRD's (Custom Resource Definitions) of the type Application into a Helm Chart. This means that you can point ArgoCD at this chart and *hey presto* it will read it's Application resources and dog-food those as new charts.

> *The Application CRD is the Kubernetes resource object representing a deployed application instance in an environment. It is defined by two key pieces of information:*
>
> `source` *reference to the desired state in Git (repository, revision, path, environment)*
>
> `destination` *reference to the target cluster and namespace.*

This paradigm of leveraging declarative resources is a powerful way to associate back to charts and/or repository sources directory and give you an out of the box experience.

You can extend this capability even further by making the bootstrap Application itself part of the initial deployment when ArgoCD is installed.

Giving a real win for simplicity and Developer Experience.


gitops-kubernetes-cluster
Kubernetes Cluster
ArgoCD Application (Custom Resource Definition)
-Sync-
API Server
helm chart git repos
ArgoCD Application (Custom Resource Definition)
HELM

Walk-through of the example

Let's walk through this repository that I use for cluster boiler plating.

If we look at the bootstrap-cluster application which can be installed at ArgoCD installation time, you'll notice that it's `repoUrl` is the same as the repository it's located in. This indicates that ArgoCD should fetch this specific chart from that repository (*why the next step will start to make more sense*).

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: bootstrap-cluster
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    helm:
      valueFiles:
      - values.yaml
    path: cluster-charts
    repoURL: https://github.com/AlexsJones/gitops-kubernetes-
cluster.git
    targetRevision: HEAD
  syncPolicy:
    automated: {}
```

Once these CRD's are applied they will automatically appear in ArgoCD as Applications with their sources.

An application as defined in cluster-charts might look like the following:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: elasticsearch
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  destination:
    namespace: elasticsearch
    server: {{ .Values.cluster }}
  project: {{ .Values.project }}
  source:
    chart: elasticsearch
    helm:
      valueFiles:
      - values.yaml
    repoURL: https://helm.elastic.co
    targetRevision: 7.6.1
```

> This is where it is important to remember because these `Applications` are being rendered via the master `cluster-chart` template they can also inherit centralised values from this chart.

In this case, I've defined in the `cluster-charts/values.yaml` two simple properties:

```
cluster: https://kubernetes.default.svc
project: default
```

Running the example

For convenience, I've wrapped most of the installation with a Makefile which you can check out to see all the commands involved.

To start with this point clone the demo repository:

This repo has a couple of extra files for ingress to my domain that we can ignore.

```
git clone https://github.com/AlexsJones/gitops-kubernetes-
cluster.git
```

Confirm you have a KubeConfig active and associated with a cluster. Typically I'd recommend starting with a local Kubernetes cluster on KIND or Minkube.

Next, we install ArgoCD

```
make install-argocd
kubectl create ns argocd || true
namespace/argocd created
kubectl apply -n argocd -f
https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
customresourcedefinition.apiextensions.k8s.io/applications.argoproj.
io created
```

```
serviceaccount/argocd-dex-server created
serviceaccount/argocd-server created
role.rbac.authorization.k8s.io/argocd-application-controller created
role.rbac.authorization.k8s.io/argocd-dex-server created
role.rbac.authorization.k8s.io/argocd-server created
clusterrole.rbac.authorization.k8s.io/argocd-application-controller
created
clusterrole.rbac.authorization.k8s.io/argocd-server created
rolebinding.rbac.authorization.k8s.io/argocd-application-controller
created
rolebinding.rbac.authorization.k8s.io/argocd-dex-server created
rolebinding.rbac.authorization.k8s.io/argocd-server created
clusterrolebinding.rbac.authorization.k8s.io/argocd-application-
controller created
clusterrolebinding.rbac.authorization.k8s.io/argocd-server created
configmap/argocd-cm created
configmap/argocd-rbac-cm created
configmap/argocd-ssh-known-hosts-cm created
configmap/argocd-tls-certs-cm created
secret/argocd-secret created
service/argocd-dex-server created
service/argocd-metrics created
service/argocd-redis created
service/argocd-repo-server created
service/argocd-server-metrics created
service/argocd-server created
deployment.apps/argocd-application-controller created
deployment.apps/argocd-dex-server created
deployment.apps/argocd-redis created
deployment.apps/argocd-repo-server created
deployment.apps/argocd-server created
```

You can use the following helping to determine when the server is ready

```
make check-argocd-ready
kubectl wait --for=condition=available deployment -l
"app.kubernetes.io/name=argocd-server" -n argocd --timeout=300s
deployment.apps/argocd-server condition met
```

Accessing ArgoCD

Retrieve the password with the Makefile command.

```
make get-argocd-password
kubectl get pods -n argocd -l app.kubernetes.io/name=argocd-server -
```

Once you have this you can login with `admin` as the username.

Connecting to the UX with a port-forward

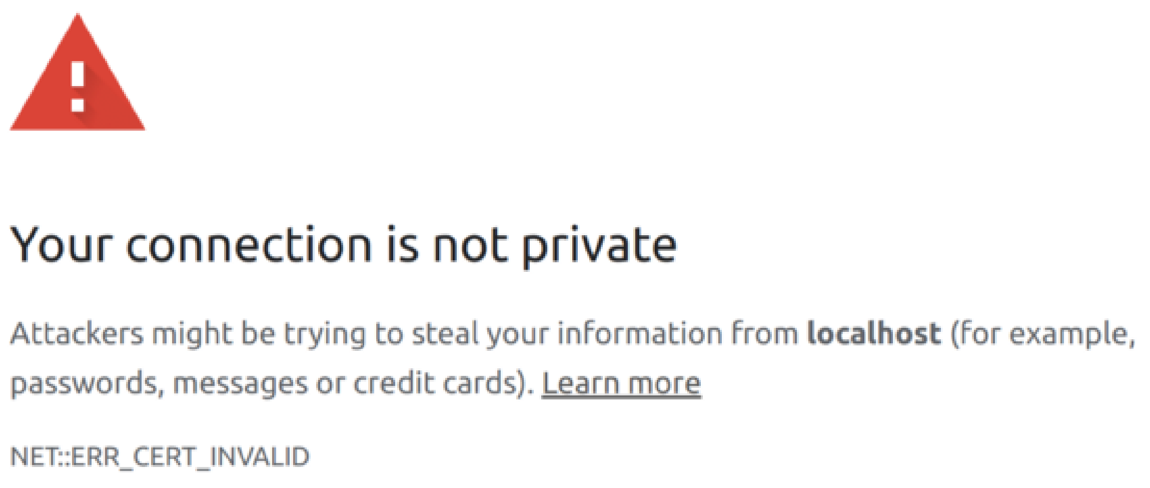
```
make proxy-argo-ui
```

Now run

```
make get-argocd-password
```

And be sure to keep a hold of this somewhere…

Note on ArgoCD HTTPS


Your connection is not private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages or credit cards). Learn more

NET::ERR_CERT_INVALID

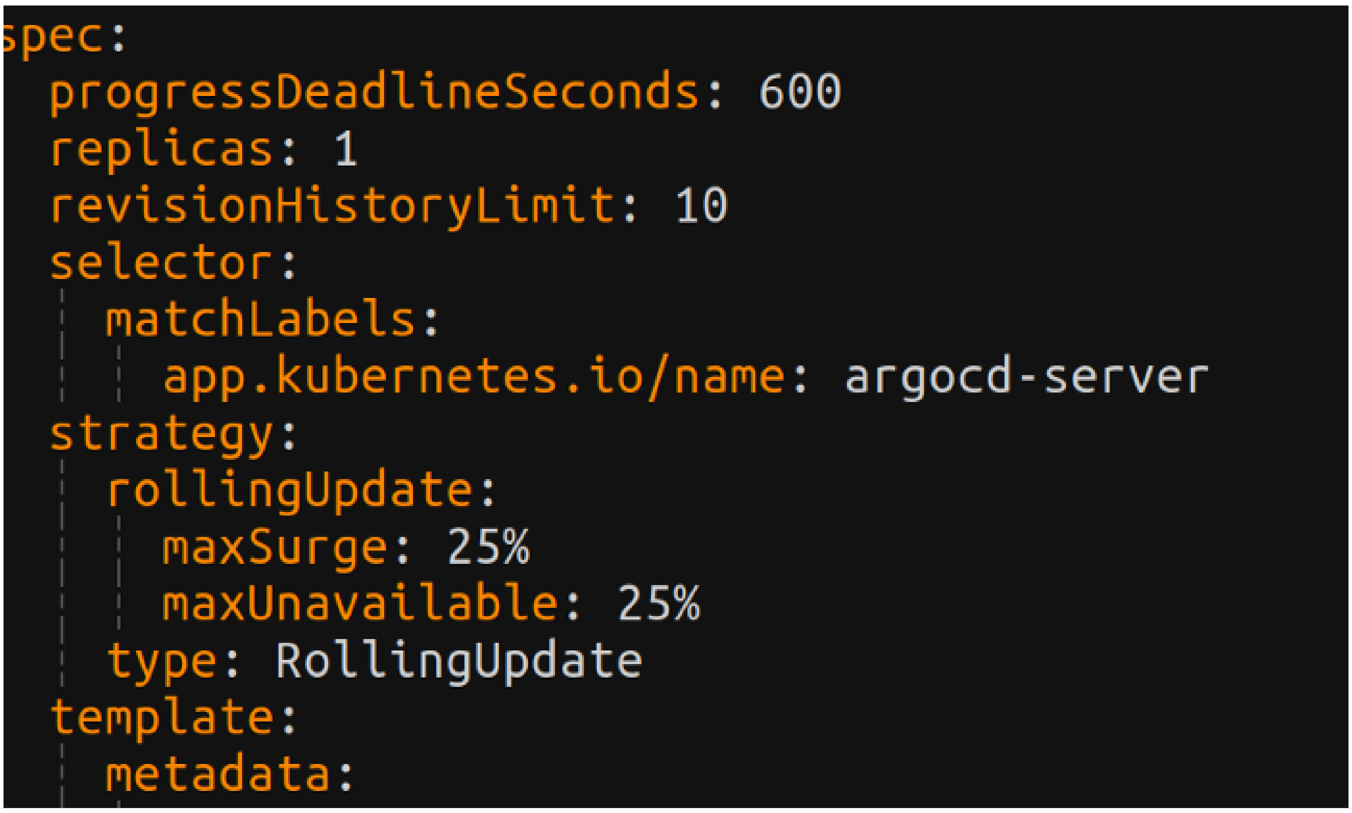
Some browsers will warn you that ArgoCD's self-signed cert is invalid if you are trying to run it locally. In addition, when you use Nginx out of the box you'll see a bunch of issues due to ArgoCD serving its own HTTPs endpoint and trying to redirect through Nginx causing a redirect loop

You can resolve this by terminating at the Nginx proxy with your domain cert and then apply this kind of Ingress resource.

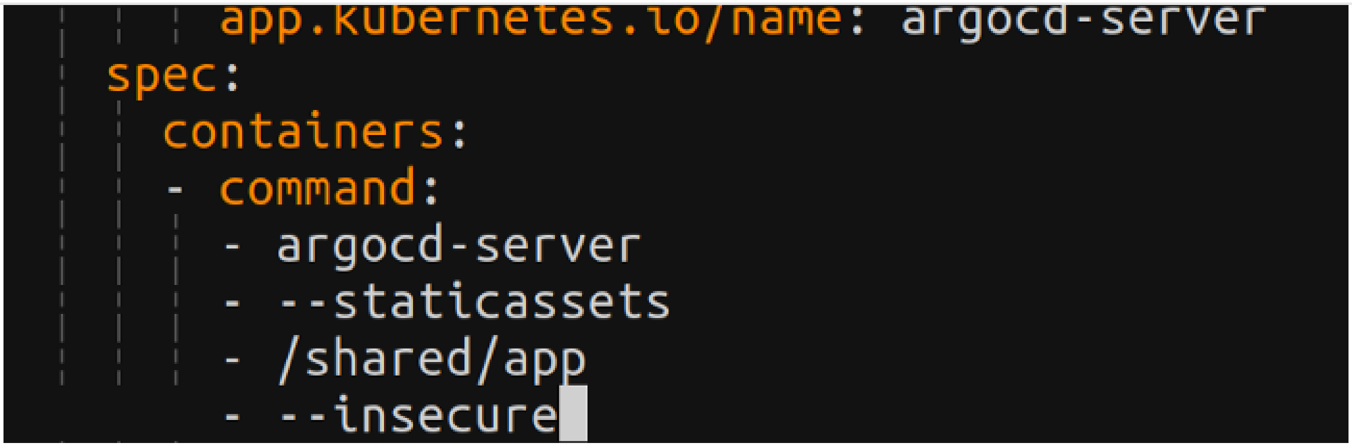
```
    name: argocd-ingress
    annotations:
      kubernetes.io/ingress.class: "nginx"
      nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
      nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
  spec:
    rules:
    - host: argocd.yourdomain.com
      http:
        paths:
        - backend:
            serviceName: argocd-server
            servicePort: http
    tls:
      - hosts:
        - argocd.iamalexsjones.com
        secretName: argocd-tls
```

> *It is important to now add the `--insecure` flag on the ArgoCD deployment with…*

```
kubectl patch deployment argocd-server --type json -p='[ { "op":
"replace",
"path":"/spec/template/spec/containers/0/command","value": ["argocd-
server","--staticassets","/shared/app","--insecure"] }]' -n argocd
```



```
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app.kubernetes.io/name: argocd-server
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
```



```
      app.kubernetes.io/name: argocd-server
  spec:
    containers:
    - command:
      - argocd-server
      - --staticassets
      - /shared/app
      - --insecure
```

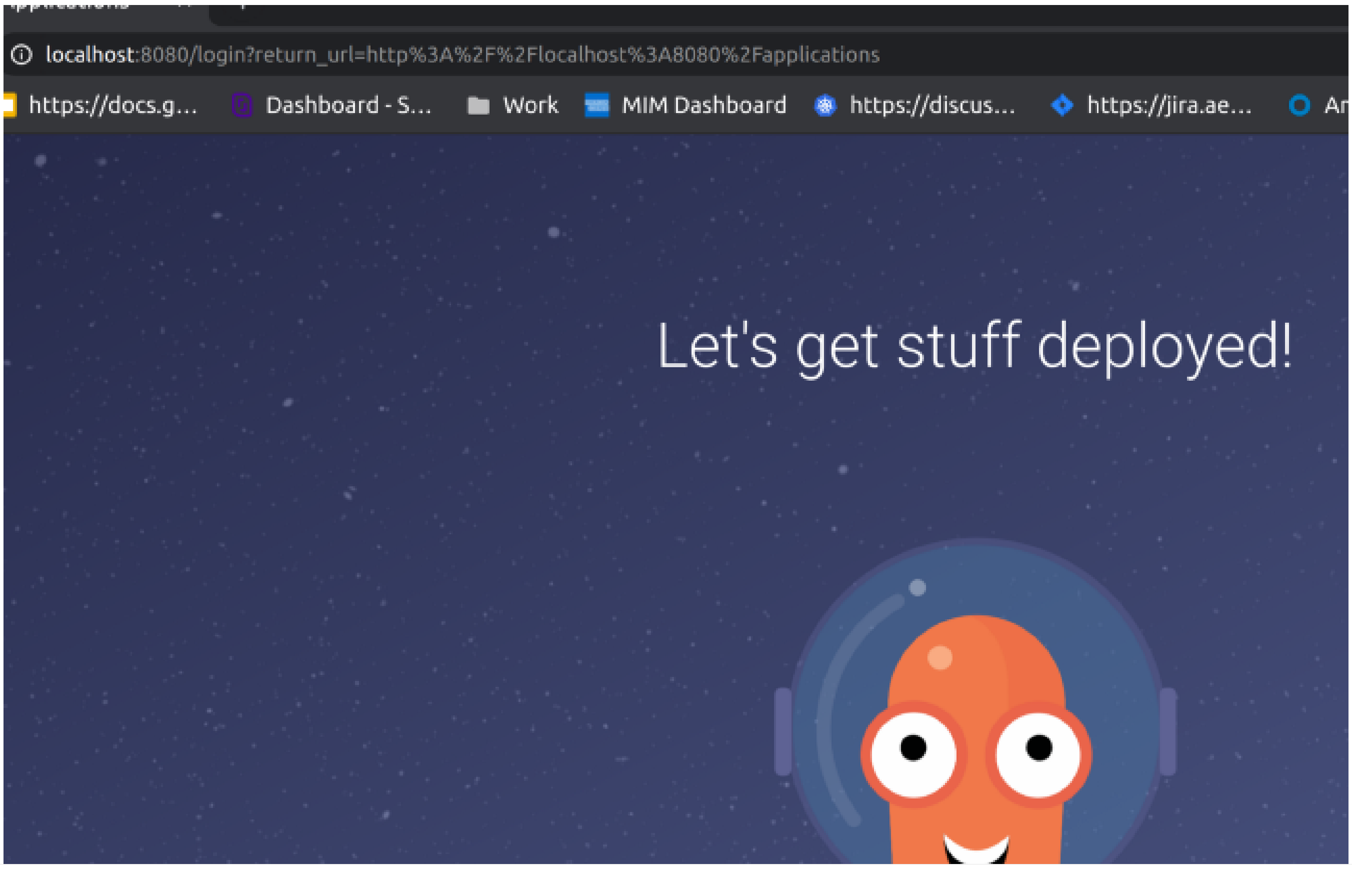
And after it restarts…

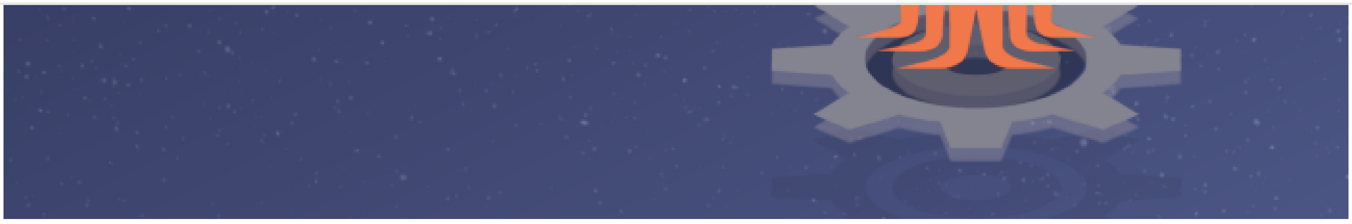
We can log-in with `` `http://localhost:8080/` `` with a port-foward

Or `https://argocd.yourdomain.com` (It using the Ingress)

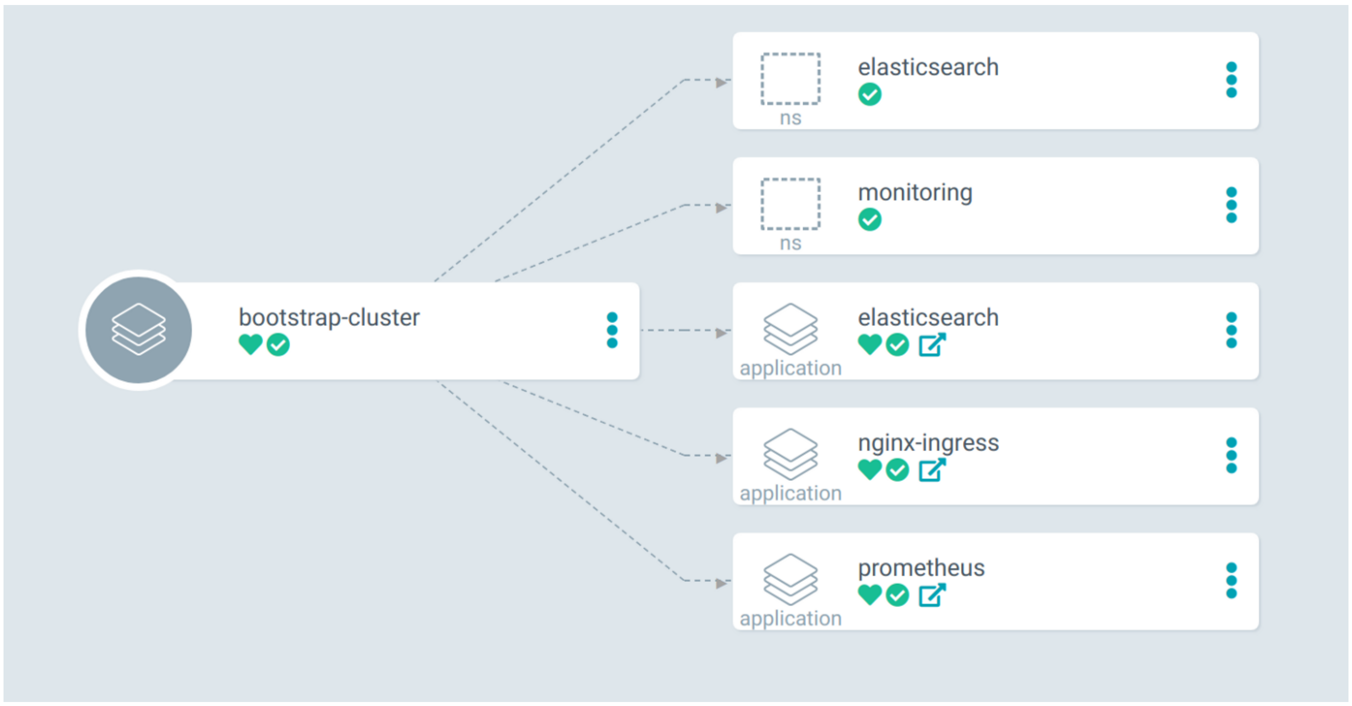
For your convenience, I added `make proxy-argcd-ui` for our demo, however, it will need the above changes made.

Note: If you find that `make get-argocd-password` is incorrect, this is because it gets set to the name of the first argo-server pod. Read this guide on how to change it.

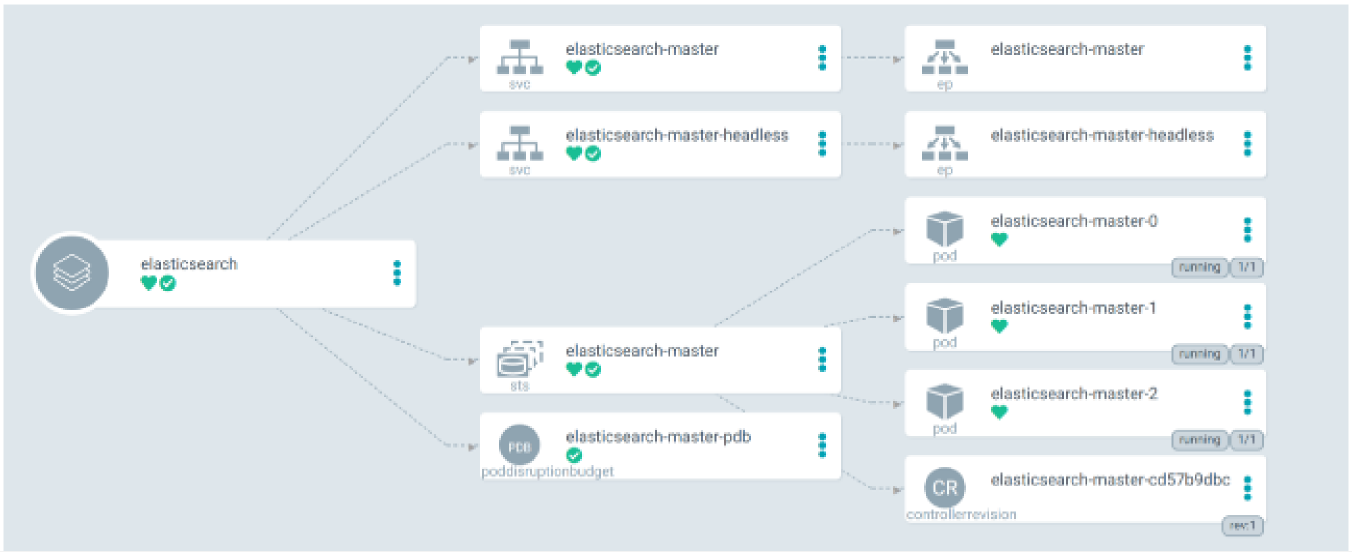

localhost:8080/login?return_url=http%3A%2F%2Flocalhost%3A8080%2Fapplications
https://docs.g...
Dashboard - S...
Work
MIM Dashboard
https://discus...
https://jira.ae...
Let's get stuff deployed!



At this point, we'll see all our deployments pre-deployed out of the box 💪



All of these Application CRD's are now rolling out.



cluster. This also means I can pin a deployment to a branch or rebuild the whole cluster if something goes wrong. 🚀

What we've learnt

- ArgoCD is a powerful GitOps platform that is completely Kubernetes Native.
- Using ArgoCD we can leverage the CRD's to use them in a declarative way to setup pipelines to Git repositories out of the box.
- ArgoCD supports multiple sources meaning our Application CRD's can describe both our Helm based chart ecosystem and out Kubernetes manifests in vanilla repositories.
- Using the Application CRD pattern we can bootstrap clusters with a big bang! 🔥

Argocd Helm Kubernetes Gitops Dx