**You need a few things.**

1. An existing Kubernetes Cluster.
2. kubectl & helm binaries locally installed

**Install Tiller (Helm server) on your cluster**

Installing Tiller is a bit more in-depth as you need to secure it in production clusters. For the purposes of keeping it simple and playing around, we will install it with normal cluster-admin roles.

If you need to secure it for a production cluster: <https://docs.helm.sh/using_helm/#tiller-and-role-based-access-control>

**Create the Tiller Service Account**

Create a folder called helm. Here we will create all Kubernetes resources for tiller. Create a file called helm/service-account.yml and add the following content:

**apiVersion**: v1  
**kind**: ServiceAccount  
**metadata**:  
 **name**: tiller  
 **namespace**: kube-system

Then apply and test that the service account exists in the cluster.

$ kubectl apply -f helm/service-account.yml  
$ kubectl get serviceaccounts -n kube-system  
NAME SECRETS AGE  
[...]  
**tiller** **1 30h**

**Create the service account role binding**

For demo purpose we will create a role binding to **cluster-admin**.

**DO NOT DO THIS IN PRODUCTION !!**

See here for more information: <https://docs.helm.sh/using_helm/#understanding-the-security-context-of-your-cluster>

Create a file called helm/role-binding.yml in the helm folder with the content:

**apiVersion**: rbac.authorization.k8s.io/v1  
**kind**: ClusterRoleBinding  
**metadata**:  
 **name**: tiller  
**roleRef**:  
 **apiGroup**: rbac.authorization.k8s.io  
 **kind**: ClusterRole  
 **name**: cluster-admin  
**subjects**:  
 - **kind**: ServiceAccount  
 **name**: tiller  
 **namespace**: kube-system

Apply and test that the role binding exists on the cluster

$ kubectl apply -f helm/role-binding.yml  
$ kubectl get clusterrolebindings.rbac.authorization.k8s.io  
NAME AGE  
[...]  
**tiller 30h**

**Deploy Tiller**

$ helm init --service-account tiller --wait

The --wait flag makes sure that tiller is finished before we apply the next few commands to start deploying Prometheus and Grafana.

Apply and test tiller is deployed and running

$ kubectl get pods -n kube-system  
NAME READY STATUS AGE  
[...]  
**tiller-deploy-dbb85cb99-st8lt** 1/1 Running 30h

**Done !**Tiller is deployed and now the real fun starts !

https://miro.medium.com/max/60/0*m1YDwiunqAsYFEMv?q=20



Photo by [Katya Austin](https://unsplash.com/@katya?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**Install Prometheus**

We will separate our monitoring resources into a separate namespace to keep them together.

Create a folder called monitoring. Here we will create all our monitoring resources.

Create a file called monitoring/namespace.yml with the content.

**kind**: Namespace  
**apiVersion**: v1  
**metadata**:  
 **name**: monitoring

Apply & Test the namespace exists.

$ kubectl get namespaces  
NAME STATUS AGE  
[...]  
**monitoring Active 105m**

**Deploy Prometheus**

Here is where the power of Helm steps in and makes life much easier.

**First we need to update our local helm chart repo.**

$ helm repo update

**Next, deploy Prometheus into the monitoring namespace**

$ helm install stable/prometheus \  
 --namespace monitoring \  
 --name prometheus

This will deploy Prometheus into your cluster in the monitoring namespace and mark the release with the name prometheus.

Prometheus is now scraping the cluster together with the node-exporter and collecting metrics from the nodes.

We can confirm by checking that the pods are running:

$ kubectl get pods -n monitoring  
NAME READY STATUS  
prometheus-alertmanager-5c5958dcb7-bq2fw 2/2 Running  
prometheus-kube-state-metrics-76d649cdf9-v5qg5 1/1 Running  
prometheus-node-exporter-j74zq 1/1 Running  
prometheus-node-exporter-x5xnq 1/1 Running  
prometheus-pushgateway-6744d69d4-27dxb 1/1 Running  
prometheus-server-669b987bcd-swcxh 2/2 Running

**Install Grafana**

When deploying grafana, we need to configure it to read metrics from the right data sources.

**Defining the grafana data sources.**

Grafana takes data sources through yaml configs when it get provisioned.   
For more information see here: <http://docs.grafana.org/administration/provisioning/#datasources>

Kubernetes has nothing to do with importing the data. Kubernetes merely orchestrates the injection of these yaml files.

When the Grafana Helm chart gets deployed, it will search for any config maps that contain a grafana\_datasource label.

**Create a Prometheus data source config map**

In the monitoring folder, create a sub-folder called grafana .

Here is where we will store our configs for the grafana deployment.

Create a file called monitoring/grafana/config.yml with the content:

**apiVersion**: v1  
**kind**: ConfigMap  
**metadata**:  
 **name**: prometheus-grafana-datasource  
 **namespace**: monitoring  
 **labels**:  
 **grafana\_datasource**: '1'  
**data**:  
 **datasource.yaml**: |-  
 apiVersion: 1  
 datasources:  
 - name: Prometheus  
 type: prometheus  
 access: proxy  
 orgId: 1  
 url: [http://prometheus-server.monitoring.svc.cluster.local](http://prometheus-server.monitoring.svc.cluster.local/)

Here is where we add the grafana\_datasource label which will tell the grafana provisioner that this is a datasource it should inject.

labels:  
 **grafana\_datasource**: '1'

Apply & test the config

$ kubectl apply -f monitoring/grafana/config.yml  
$ kubectl get configmaps -n monitoring  
NAME DATA AGE  
[...]  
**grafana 1 131m**

**Override Grafana value**

When Grafana gets deployed and the provisioner runs, the data source provisioner is deactivated. We need to activate it so it searches for our config maps.

We need to create our own values.yml file to override the datasources search value, so when Grafana is deployed it will search our datasource.yml definition and inject it.

Create a file called monitoring/grafana/values.yml with the content:

**sidecar**:  
 **image**: xuxinkun/k8s-sidecar:0.0.7  
 **imagePullPolicy**: IfNotPresent  
 **datasources**:  
 **enabled**: true  
 **label**: grafana\_datasource

This will inject a sidecar which will load all the data sources into Grafana when it gets provisioned.

Now we can deploy Grafana with the overridden values.yml file and our datasource will be imported.

$ helm install stable/grafana \  
 -f monitoring/grafana/values.yml \  
 --namespace monitoring \   
 --name grafana

Check that it is running:

$ kubectl get pods -n monitoring  
NAME READY STATUS RESTARTS AGE  
**[...]  
grafana-5f4d8bcb94-ppsjq 1/1 Running**

**Get the Grafana Password**

Grafana is deployed with a password. This is good news. But whats the password ?

$ kubectl get secret \  
 --namespace monitoring grafana \  
 -o jsonpath="{.data.admin-password}" \  
 | base64 --decode ; echo

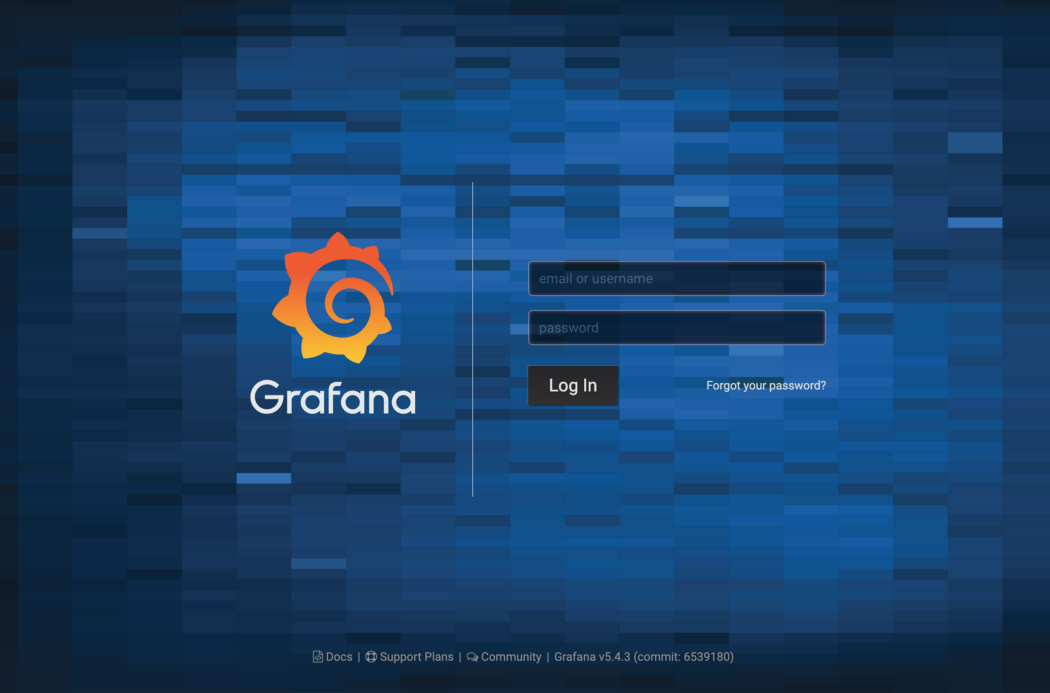
This will spit out the password to your Grafana dashboard.   
The username is admin

Port Forward the Grafana dashboard to see whats happening:

$ export POD\_NAME=$(kubectl get pods --namespace monitoring -l "app=grafana,release=grafana" -o jsonpath="{.items[0].metadata.name}")  
$ kubectl --namespace monitoring port-forward $POD\_NAME 3000

Go to [http://localhost:3000](http://localhost:3000/) in your browser. You should see the Grafana login screen:

https://miro.medium.com/max/60/1*eWD53Uk1hLzcGj9HGjOPww.png?q=20



Smashing login screen

Login with the username and password you have from the previous command.

**Add a dashboard**

Grafana has a long list of prebuilt dashboard here:   
<https://grafana.com/dashboards>

Here you will find many many dashboards to use. We will use [this one](https://grafana.com/dashboards/1860) as it is quite comprehensive in everything it tracks.

In the left hand menu, choose Dashboards > Manage > + Import