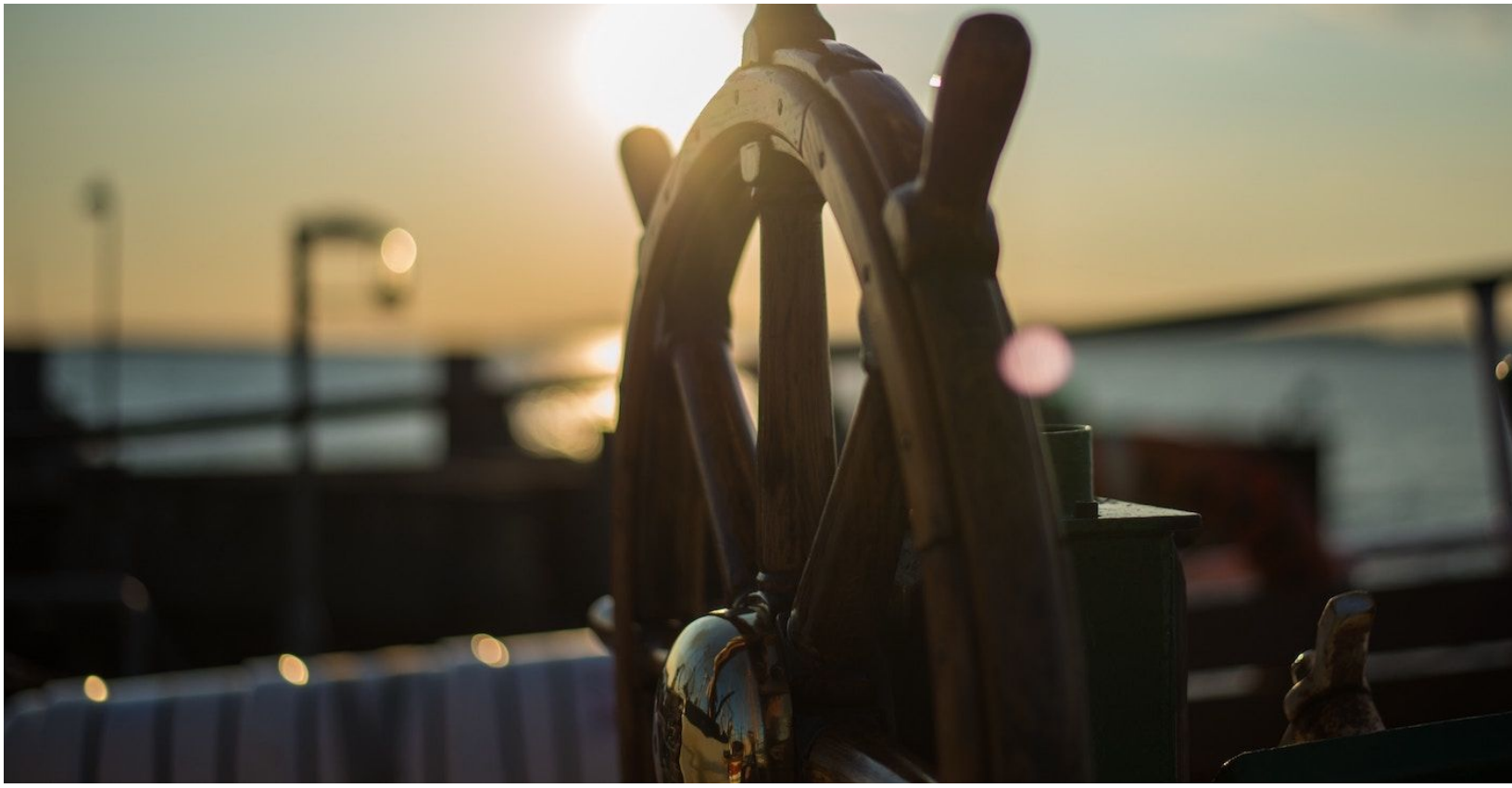
Kubeapps, the Kubernetes Dashboard

One-click to Deploy and Manage Helm Charts on Kubernetes

Index



Kubeapps, the Kubernetes dashboard

Kubeapps is a web application designed for deploying and managing applications in Kubernetes clusters. **Unlike the Kubernetes dashboard, Kubeapps provides a central location for your applications and their full life-cycle.** Using it, your cluster users can deploy applications packaged as Helm charts directly from their browsers. Continue reading to discover its main features and how you, as a cluster operator, can benefit from managing your deployments from a single and centralized user interface.

What does Kubeapps offer you in essence?

In summary, Kubeapps allows you to:

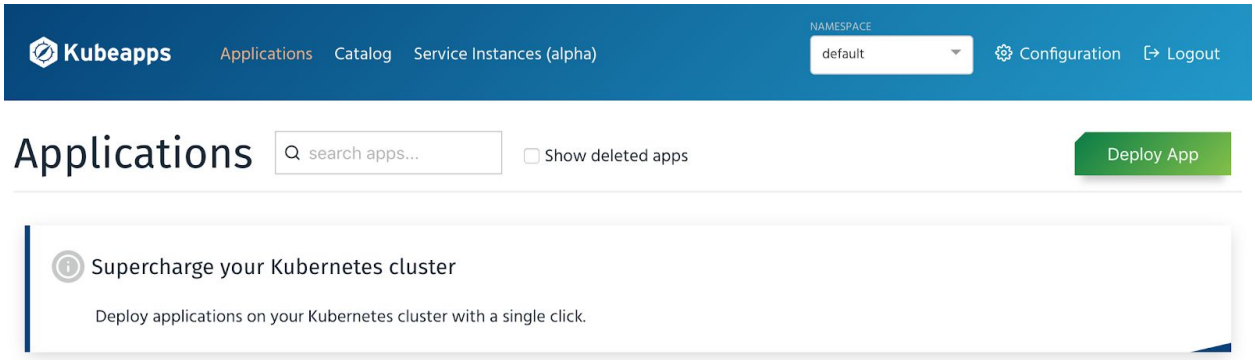
- Browse and deploy Helm charts from chart repositories
- Inspect, upgrade and delete Helm-based applications installed in the cluster

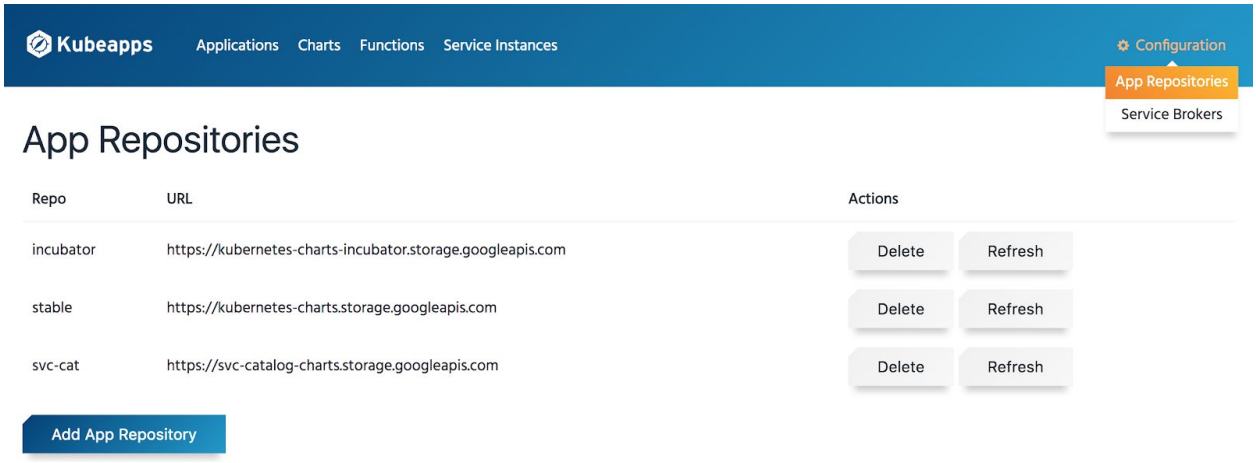
- Add custom and private chart repositories (supports ChartMuseum and JFrog Artifactory)
- Browse and provision external services from the Service Catalog and available Service Brokers
- Connect Helm-based applications to external services with Service Catalog Bindings
- Secure authentication and authorization based on Kubernetes Role-Based Access Control

Deploy and Manage Trusted Applications

Kubeapps includes a built-in catalog with numerous Helm charts packaged and updated by Bitnami. You can also add your own private repository (supports ChartMuseum and JFrog Artifactory), so you can deliver trusted applications packaged following your company's best practices to your users.

Both ways ensure that **anyone who wants to run workloads in your Kubernetes cluster** or clusters **should only choose from applications that your IT department has selected and packaged**, and include the latest version by default. This is something that cluster admins will find very valuable for simplifying the manage of compliance and governance in your environment.

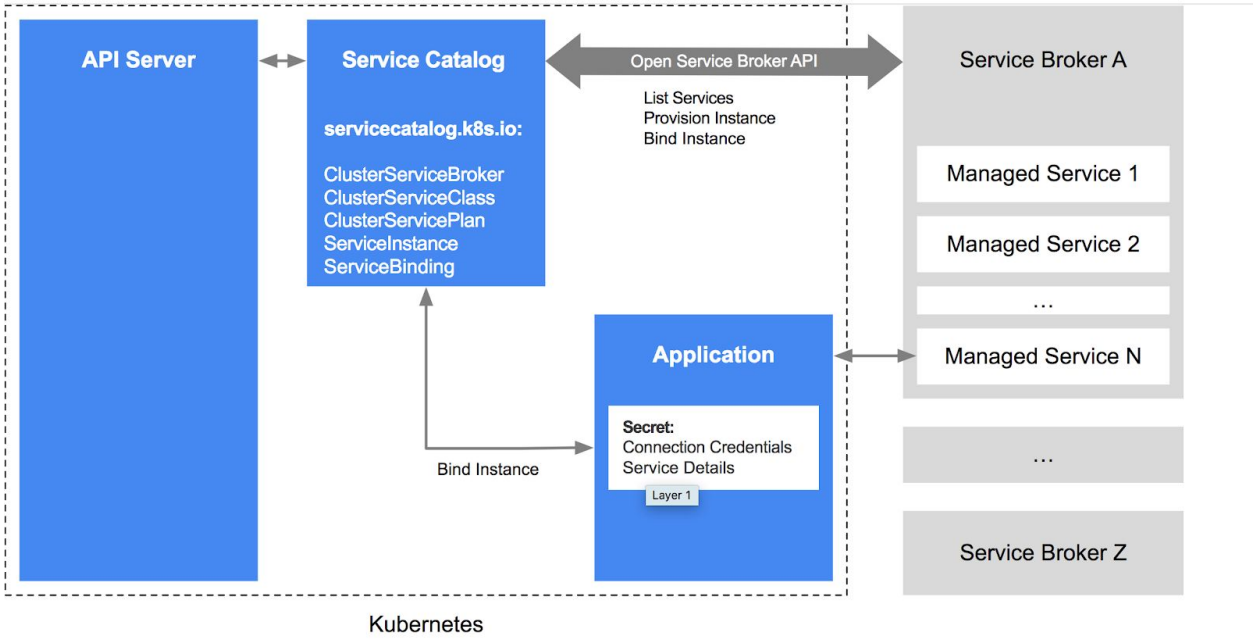




Not only can Kubeapps be used for deploying applications, as mentioned, **you can manage the entire life cycle of your applications from its dashboard.** The system enables you to update an existing deployment when an updated version is available. You can also delete unused or outdated applications that are deployed in your Kubernetes cluster.

Provision External Services from the Service Catalog

Many applications depend on managed services that are only offered by cloud providers. Developers sometimes prefer to delegate the management of some services such as databases or storage to a public cloud. Thus, they can focus on deploying their applications to a Kubernetes cluster without the worry of creating backups, enabling high-availability, etc. of those services by themselves.



Service Catalog Architecture - kubernetes.io

But all of those external services need to be connected to their applications in some way. **The Service Catalog acts as a tool for instantiate services outside Kubernetes directly from a Kubernetes cluster in a full declarative way.** It is an extension API that enables applications running on Kubernetes to use external managed software offerings.

Kubeapps allows you to browse and provision, directly from its UI, external services from the Kubernetes Service Catalog and Service Brokers.

Secure Access to Your Kubernetes Cluster

Kubeapps requires users to login with a Kubernetes API token in order to make requests to the Kubernetes API server as the user. This ensures that a certain user of Kubeapps is only permitted to view and manage applications that they have access to (for example, within a specific namespace). If a user does not have access to a particular resource, Kubeapps will display an error describing the required roles to access the resource.

If your cluster supports Token Authentication you may login with the same token.

Designed for Cluster Administrators and Operators

Now that you have a complete overview of what is Kubeapps and how it can help you to operate securely your cluster, it is time to start using it to operate your Kubernetes clusters.

You can use Kubeapps standalone or combined with other Bitnami products and Kubernetes projects such as Stacksmith or BKPR. Furthermore, Kubeapps enables you to provision your cluster with external cloud services from Kubernetes by adding the Kubernetes Service Catalog. That way, you will be able to use external managed software offerings from the major cloud vendors and software providers directly on your cluster.

In the pages of this e-book, you will find out how to use and adopt Kubeapps with ease. The topics covered in this e-book have been collected both from the Kubeapps documentation as well as articles published in the Bitnami Engineering Portal. Topics included are:

- Installing Kubeapps
- Deploy and Manage Applications with the Kubeapps UI
- How Kubeapps Enforces the Security of Your Kubernetes Deployments
- Kubernetes Service Catalog Kubeapps Integration

Installing Kubeapps

Get started with Kubeapps

Now that you have learnt how Kubeapps helps you deploy securely Helm charts on your Kubernetes cluster, it is time to take action and test it in your cluster. Follow these instructions to install Kubeapps in your cluster and deploy a sample application.

Prerequisites

To install the latest version of Kubeapps you need to have:

- A Kubernetes cluster running (v1.8+)
- Helm installed in your cluster (v2.10.0+)
- `Kubectl` CLI installed and configured

Compatible Kubernetes Services and Engines

Kubeapps can be used with the following Kubernetes services:

- Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)
- Oracle Container Engine for Kubernetes (OKE)
- Minikube
- Docker for Desktop Kubernetes

Step 1: Install Kubeapps

Use the Helm chart to install the latest version of Kubeapps:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install --name kubeapps --namespace kubeapps bitnami/kubeapps
```

The above commands will deploy Kubeapps into the `kubeapps` namespace in your cluster. It may take a few minutes to execute. Wait until it is completely deployed and the Kubeapps pods are running.

Step 2: Create a Kubernetes API token

The access to the Dashboard requires a Kubernetes API token to authenticate with the Kubernetes API server.

```
kubectl create serviceaccount kubeapps-operator
kubectl create clusterrolebinding kubeapps-operator
--clusterrole=cluster-admin
--serviceaccount=default:kubeapps-operator
```

NOTE: It is not recommended to create `cluster-admin` users for Kubeapps production usage.

Retrieve the Token

Depending on the operating system you are using you will need to perform different actions to retrieve the token:

On Linux / macOS

Execute this command:

```
kubectl get secret $(kubectl get serviceaccount kubeapps-operator -o
jsonpath='{.secrets[].name}') -o jsonpath='{.data.token}' | base64
--decode
```

On Windows

Create a file called `GetDashToken.cmd` with the following lines in it:

```
@ECHO OFF
REM Get the Service Account
kubectl get serviceaccount kubeapps-operator -o
jsonpath={.secrets[].name} > s.txt
SET /p ks=<s.txt
DEL s.txt

REM Get the Base64 encoded token
kubectl get secret %ks% -o jsonpath={.data.token} > b64.txt

REM Decode The Token
DEL token.txt
certutil -decode b64.txt token.txt
```

Open a command prompt and run the `GetDashToken.cmd` Your token can be found in the `token.txt` file.

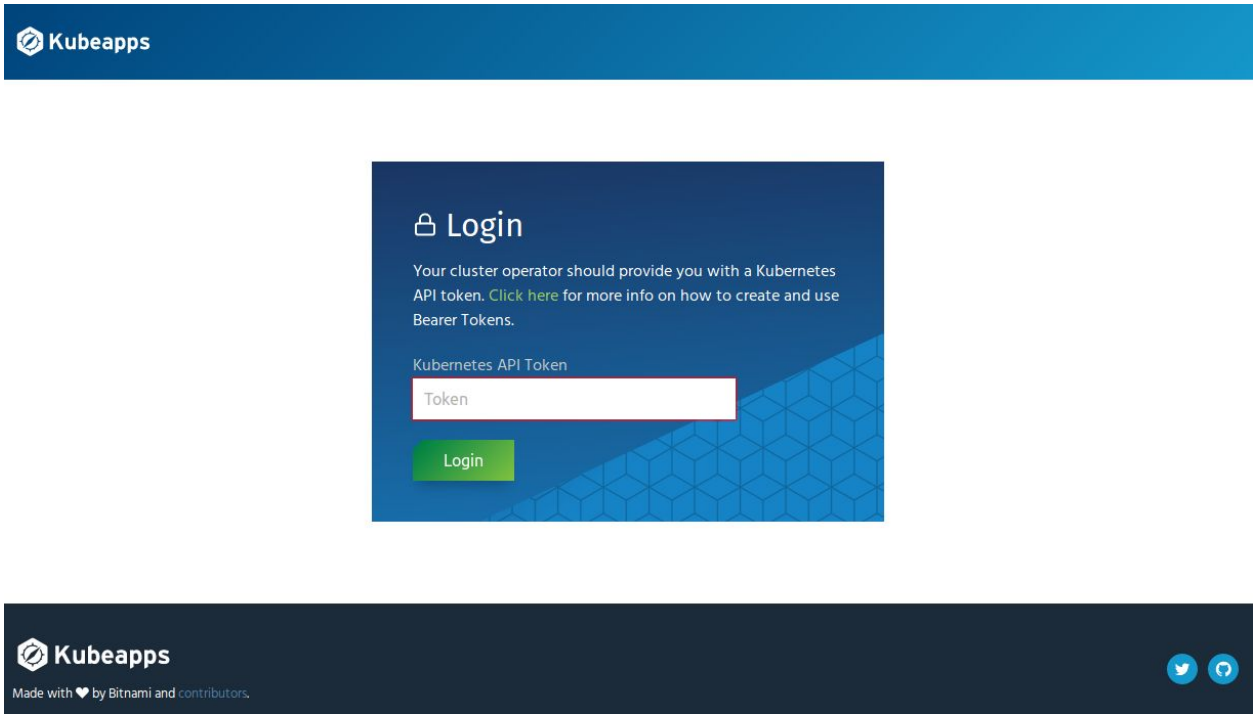
Are you an Oracle Container Engine for Kubernetes user? Find out how to deploy Kubeapps in a cluster running on OKE.

Step 3: Start the Kubeapps Dashboard

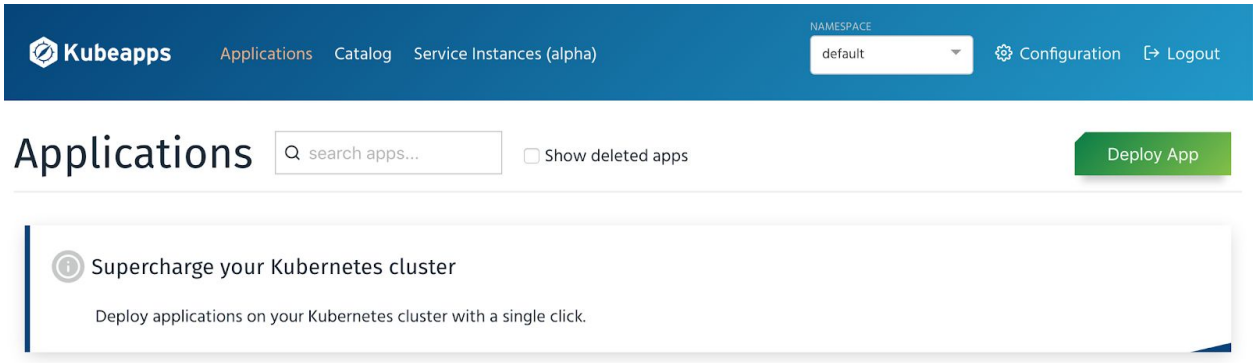
Once Kubeapps is installed, securely access the Kubeapps Dashboard from your system by running:

```
export POD_NAME=$(kubectl get pods -n kubeapps -l
"app=kubeapps,release=kubeapps" -o
jsonpath="{.items[0].metadata.name}")
echo "Visit http://127.0.0.1:8080 in your browser to access the
Kubeapps Dashboard"
kubectl port-forward -n kubeapps $POD_NAME 8080:8080
```

This will start an HTTP proxy for secure access to the Kubeapps Dashboard. Visit *http://127.0.0.1:8080/* in your preferred web browser to open the Dashboard. Here's what you should see:



Paste the token generated in the previous step to authenticate and access the Kubeapps dashboard for Kubernetes.

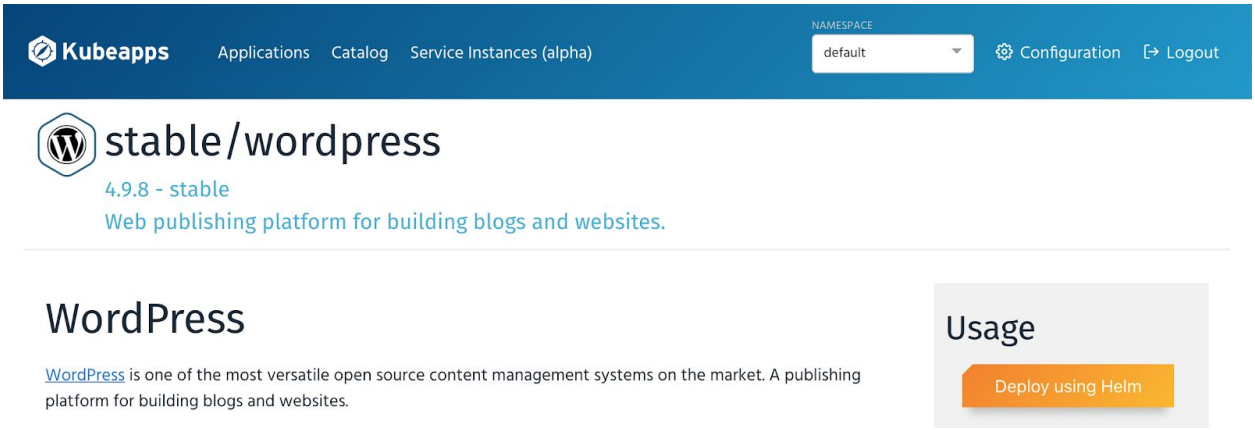


Step 4: Deploy WordPress

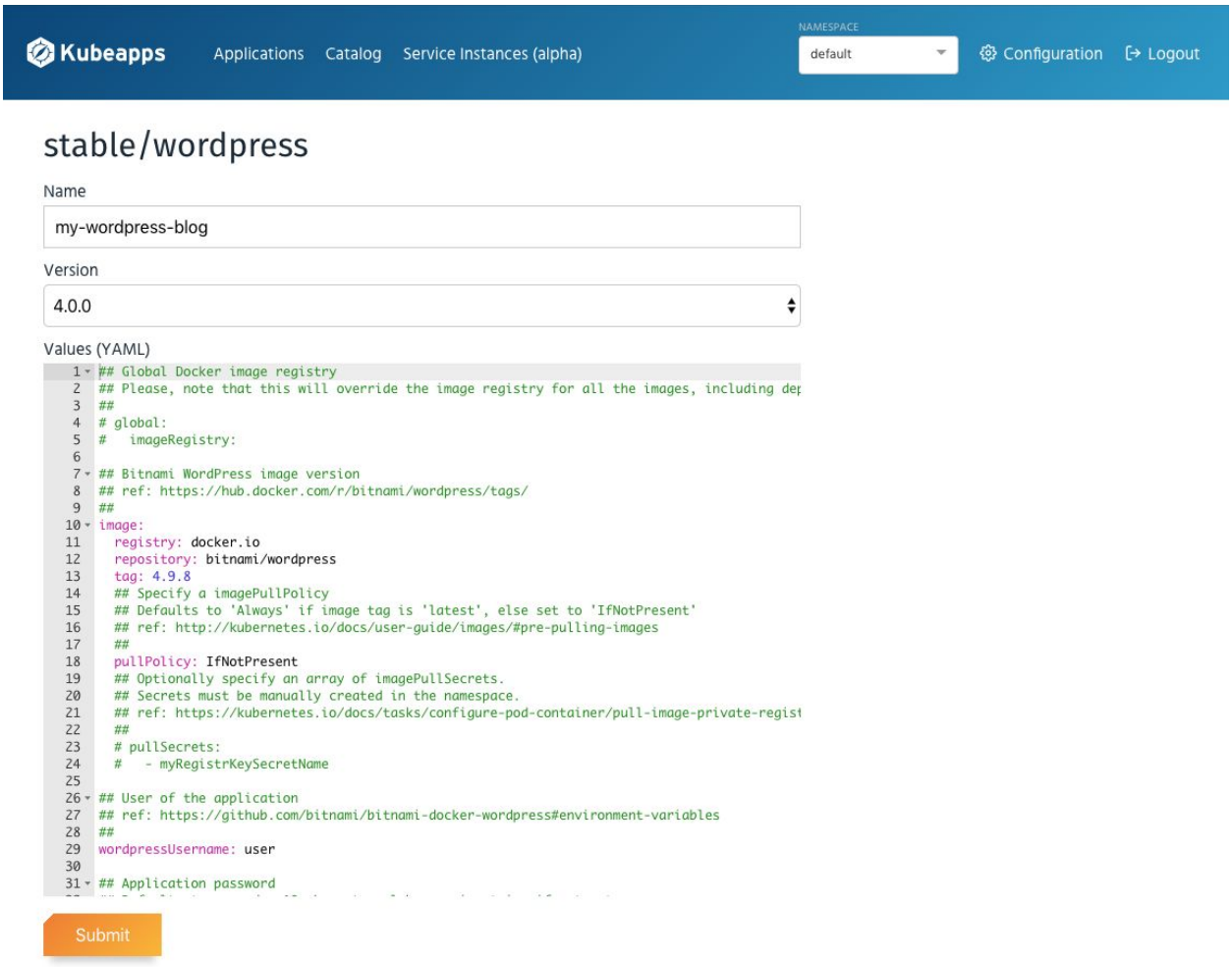
Once you have the Kubeapps Dashboard up and running, you can start deploying applications into your cluster.

- Use the "Catalog" page in the Dashboard to select an application from the list of charts in any of the configured Helm chart repositories. This example assumes you want to deploy WordPress.

- Click the "Deploy using Helm" button.

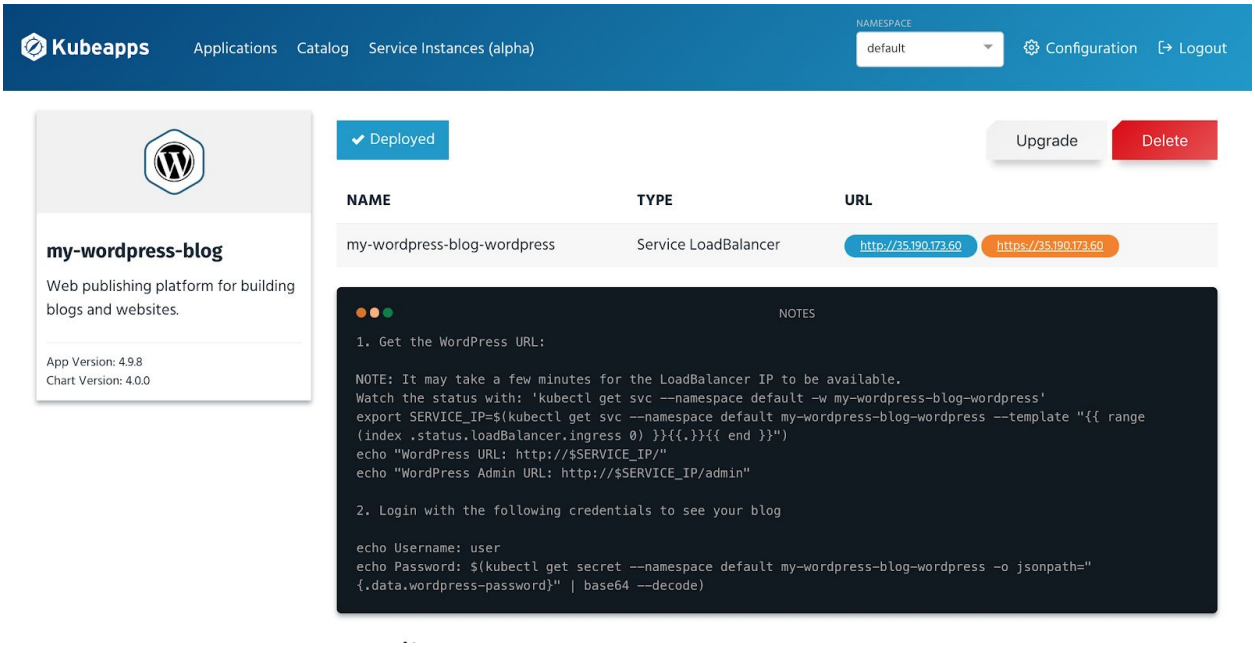


- You will be prompted for the release name and values for the application.



- Click the "Submit" button. The application will be deployed. You will be able to track the new Kubernetes deployment directly from the browser.

To obtain the WordPress username and password, refer to the "Notes" section of the deployment page, which contains the commands you will need to run to obtain the credentials for the deployment.



You can also use the URLs shown to directly access the application. Note that, depending on your cloud provider of choice, it may take some time for an access URL to be available for the application and the Service will stay in a "Pending" state until a URL is assigned. If using Minikube, you will need to run `minikube tunnel` in your terminal in order for an IP address to be assigned to your application.

```
NOTES
1. Get the WordPress URL:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.
Watch the status with: 'kubectl get svc --namespace default -w default-my-wordpress-blog-wordpress'

export SERVICE_IP=$(kubectl get svc --namespace default default-my-wordpress-blog-wordpress -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
echo http://$SERVICE_IP/admin

2. Login with the following credentials to see your blog

echo Username: user
echo Password: $(kubectl get secret --namespace default default-my-wordpress-blog-wordpress -o jsonpath="
{.data.wordpress-password}" | base64 --decode)
```

Deploy and Manage Applications with the Kubeapps UI

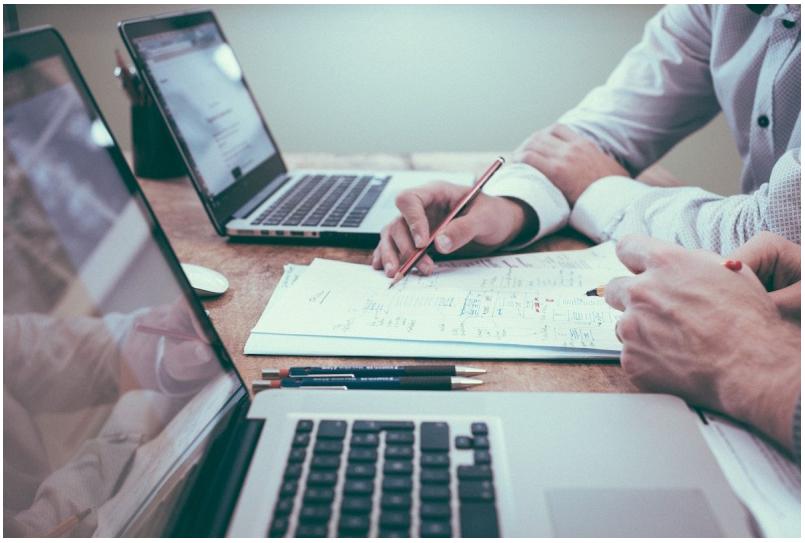
How to Deploy Your Custom Application Using Kubeapps

Authored by Carlos R. Hernandez, Software Engineer
April 22, 2019

Bitnami is continually investing in tools to address the next generation of application packaging including innovations like containers and serverless computing. Kubernetes is a prominent tool for container orchestration while Bitnami is leading the way that containers and functions are packaged and delivered for that platform.

In this regard, Bitnami also provides supporting Open Source tools for Kubernetes users such as: Bitnami Kubernetes Production Runtime (BKPR), Kubeless, Kubecfg, Kubewatch, or Sealed Secrets. And last but not least, the Bitnami key project for Kubernetes: Kubeapps.

Kubeapps is a web-based UI for deploying and managing applications in Kubernetes clusters. It includes a built-in catalog of Helm charts that you can deploy on your Kubernetes cluster, and allows you to manage, upgrade, and delete them directly from its dashboard. You can also use it to **connect your deployments to external services from the Kubernetes Service Catalog, as well as to secure them with Kubernetes RBAC rules.**

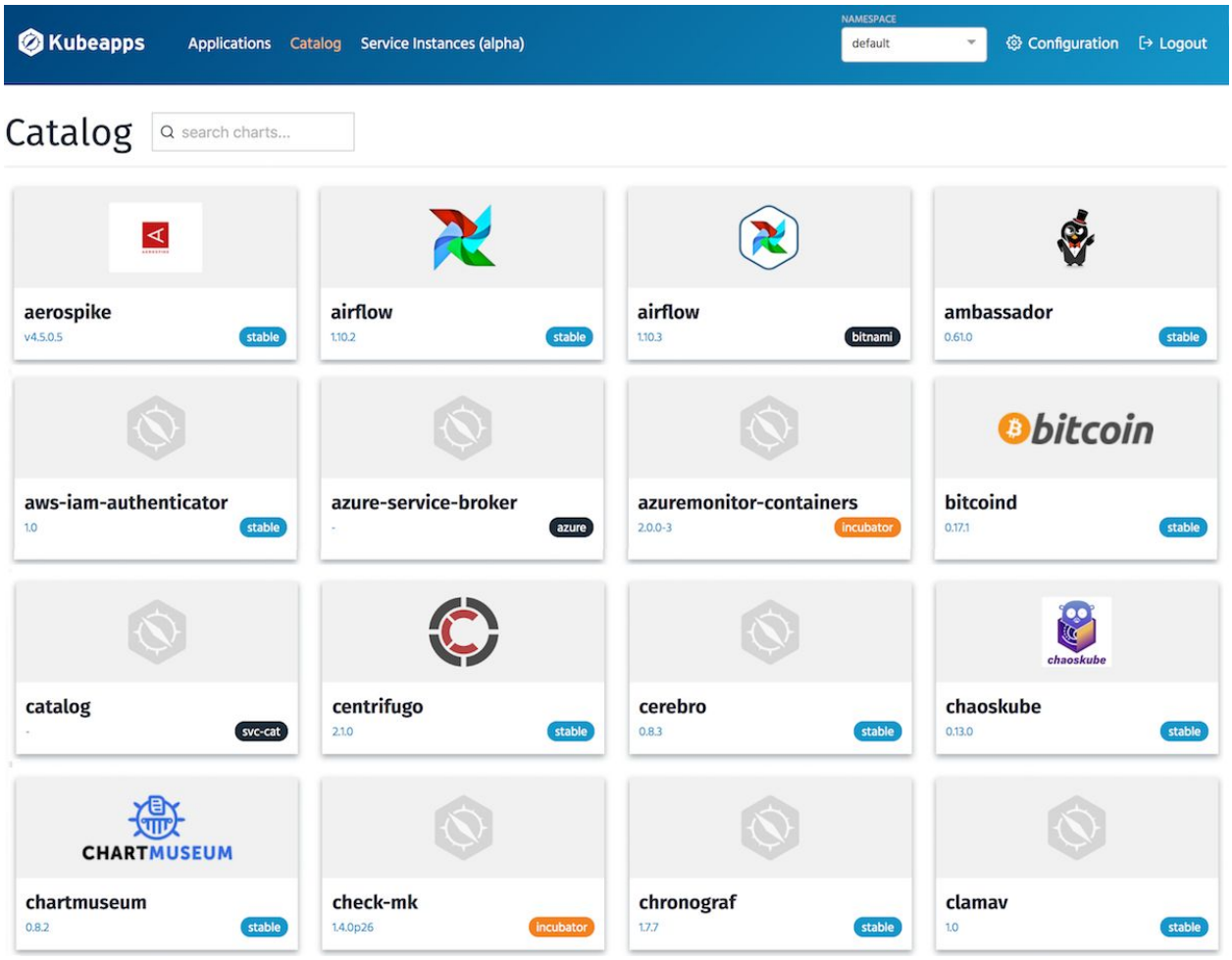


With Kubeapps, you can add your own chart repo so you can deliver only trusted applications to your cluster users. This blog post will walk you through the process of adding your own repo to Kubeapps.

After that, it will show you how to deploy a chart from the Kubeapps UI.To do so, a Java Tomcat application with MySQL, that has previously been packaged with Bitnami Stacksmith, will be used as an example, but you can use any of the charts included in the Kubeapps catalog or one from your own Helm chart repository.

How To Create Your Own Chart Repository On Kubeapps

As mentioned above, Kubeapps comes with different chart repositories that are enabled. See the list of enabled chart repositories in the "Catalog" section:



Furthermore, it is possible to use a private Helm repository to store your own Helm charts (providing your cluster uses only trusted applications that accomplish your internal policies

and best practices) and to deploy them using Kubeapps. The following instructions show how to install Kubeapps for creating your own chart repository using ChartMuseum. These are the steps to follow:

- Install Kubeapps
    - Create a Kubernetes API token
    - Start the Kubeapps Dashboard
- Download the ChartMuseum chart from the Kubeapps catalog
- Upload a chart to ChartMuseum
- Configure your chart repository in Kubeapps
- Configure Authentication/Authorization

Install Kubeapps

Use the Helm chart to install the latest version of Kubeapps:

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm install --name kubeapps --namespace kubeapps bitnami/kubeapps
```

The commands above will deploy Kubeapps into the *kubeapps* namespace on your cluster. It may take a few minutes to execute. Once it has been deployed and the Kubeapps pods are running, continue to step two.

Create A Kubernetes API Token

Access to the dashboard requires a Kubernetes API token to authenticate with the Kubernetes API server.

```
$ kubectl create serviceaccount kubeapps-operator
$ kubectl create clusterrolebinding kubeapps-operator
--clusterrole=cluster-admin
--serviceaccount=default:kubeapps-operator
```

To retrieve the token, execute the following:

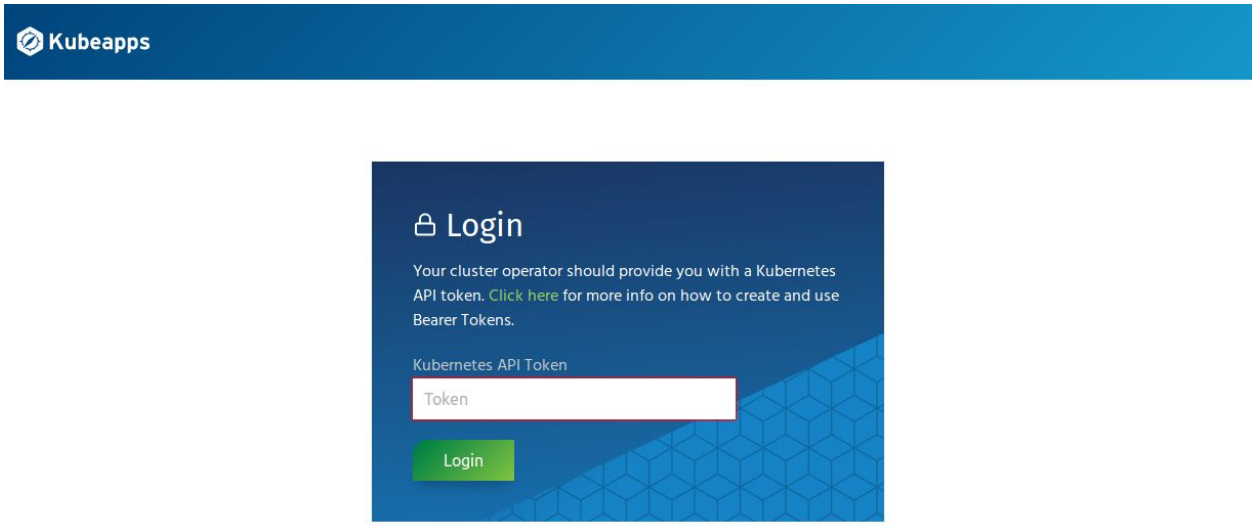
```
$ kubectl get secret $(kubectl get serviceaccount kubeapps-operator
-o jsonpath='{.secrets[].name}') -o jsonpath='{.data.token}' | base64
--decode
```

Start The Kubeapps Dashboard

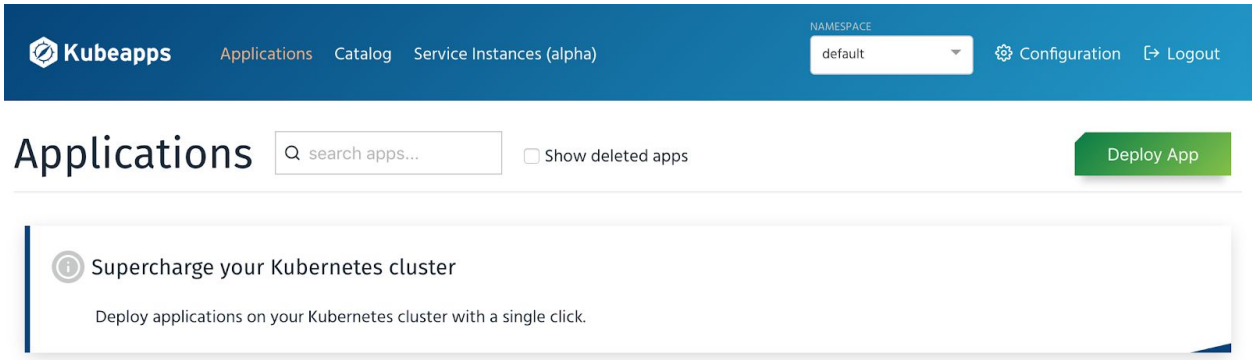
Once Kubeapps is installed, to securely access the Kubeapps Dashboard from your system, run the following:

```
$ export POD_NAME=$(kubectl get pods -n kubeapps -l
"app=kubeapps,release=kubeapps" -o
jsonpath="{.items[0].metadata.name}")
$ echo "Visit http://127.0.0.1:8080 in your browser to access the
Kubeapps Dashboard"
$ kubectl port-forward -n kubeapps $POD_NAME 8080:8080
```

This will start an HTTP proxy for secure access to the Kubeapps Dashboard. Visit http://127.0.0.1:8080/ in your preferred web browser to open the dashboard. Here's what you should see:



- Paste the token generated in the previous step to authenticate and access the Kubeapps Dashboard for Kubernetes.



Do you want to package your own applications and deploy them from the Kubeapps UI? Read this set of articles about [how to create a trusted set of applications with Stacksmith and Kubeapps.]

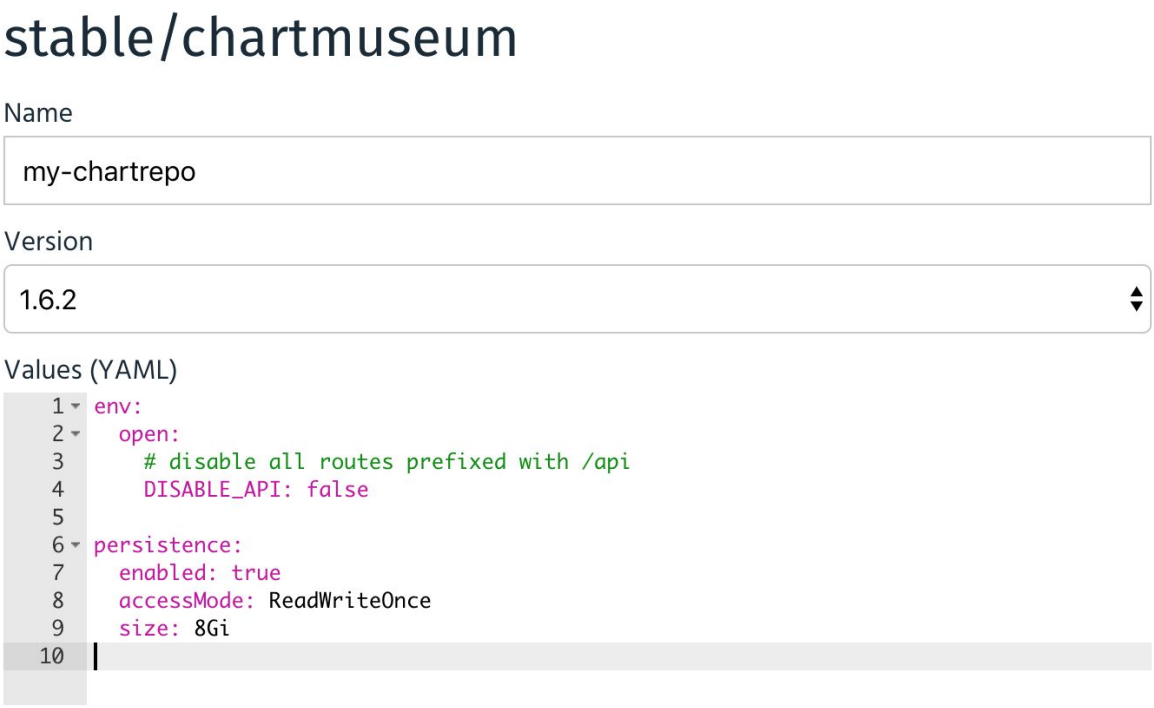
Download The ChartMuseum Chart From The Kubeapps Catalog

Now that you have Kubeapps installed and ready to use, it is time to start using ChartMuseum so you will be able to create your own private repository.

To use ChartMuseum with Kubeapps, it is necessary to deploy a Helm chart from the Kubeapps catalog. Browse through it and select its *stable* repository:

Make the following changes in the values.yml file:

- env.open.DISABLE_API: Set this value to *false* to be able to use the ChartMuseum API to push new charts.
- persistence.enabled: Set this value to *true* to enable persistence for the charts you store. Note that this will create a Kubernetes Persistent Volume Claim so depending on your Kubernetes provider, you may need to manually allocate the required Persistent Volume (PV) to satisfy the claim. Some Kubernetes providers automatically create PVs for you, so setting this value to *true* is enough.



Upload A Chart To ChartMuseum

Once ChartMuseum is deployed on your cluster, you will be able to upload a chart. To open a terminal and create a port-forward tunnel to the application, execute the following:

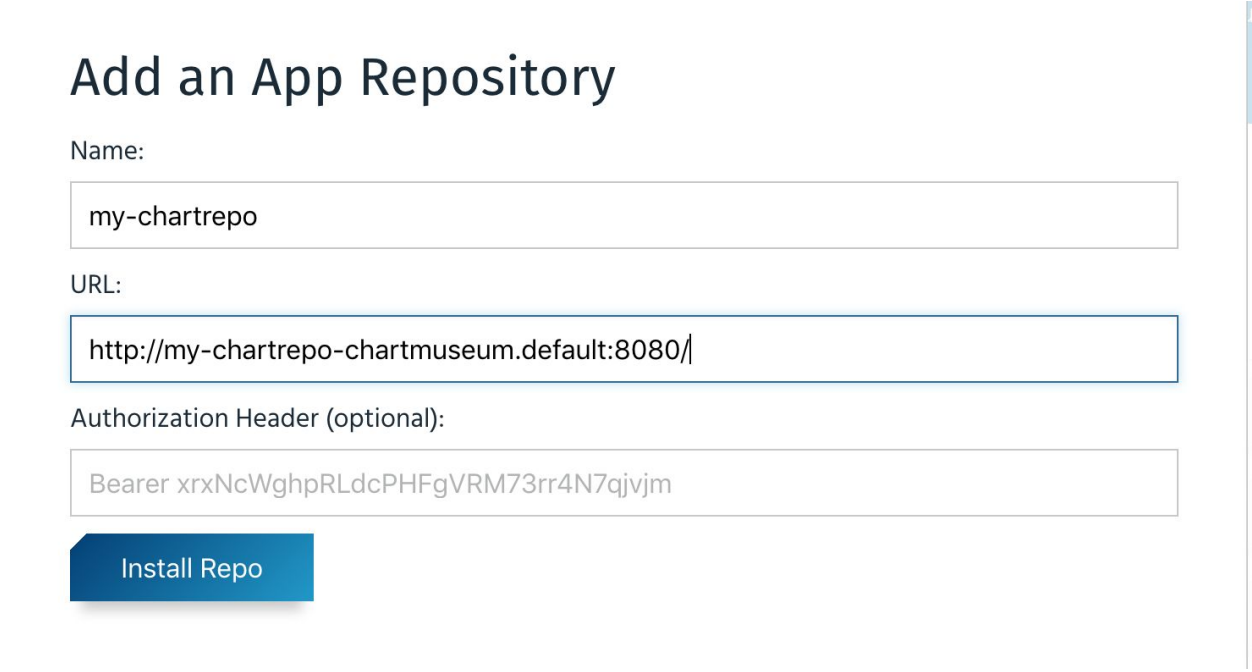
```
$ export POD_NAME=$(kubectl get pods --namespace default -l
"app=chartmuseum" -l "release=my-chartrepo" -o
jsonpath="{.items[0].metadata.name}")
$ kubectl port-forward $POD_NAME 8080:8080 --namespace default
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

And in a different terminal, you can push your chart:

```
$ helm package /path/to/my/chart
Successfully packaged chart and saved it to:
/path/to/my/chart/my-chart-1.0.0.tgz
$ curl --data-binary "@my-chart-1.0.0.tgz"
http://localhost:8080/api/charts
{"saved":true}
```

Configure Your Chart Repository In Kubeapps

To add your private repository, go back to the Kubeapps UI and navigate to *Configuration -> App Repositories* and click on "Add App Repository." Use the Kubernetes DNS name for the ChartMuseum service. This will be `<release_name>-chartmuseum.<namespace>:8080`:



Once you create the repository, click on the link of the repository you want to use and start to deploy your own applications using Kubeapps.

Configure Authentication/Authorization

Authentication/authorization mechanisms which are used to identify the user have many advantages. The main purpose of these mechanisms is to validate the user's right to access the information and protect against identity theft and fraud. It is possible to configure ChartMuseum to use authentication with two different mechanisms:

1) Using HTTP basic authentication (user/password). To use this feature, you must:

- Specify the parameters `secret.AUTH_USER` and `secret.AUTH_PASS` when deploying the ChartMuseum.
- Set as Authorization Header a *Basic* authorization header using the base64 codification of the *user:password* string as the value. For example, for the user "JaneDoe" and password "secretPassword" (`echo "JaneDoe:secretPassword" | base64`), it would use the following header field: `Basic SmFuZURvZTpzZWNyZXRQYXNzd29yZAo=`

2) Using a JWT token. Once you obtain a valid token, set it in the Authorization Header field of the App Repository form. Note that in this case, it will be prefixed with *Bearer*. For example: *Bearer UVd4aFpHUnBianB2Y0dWdUlIT=*.

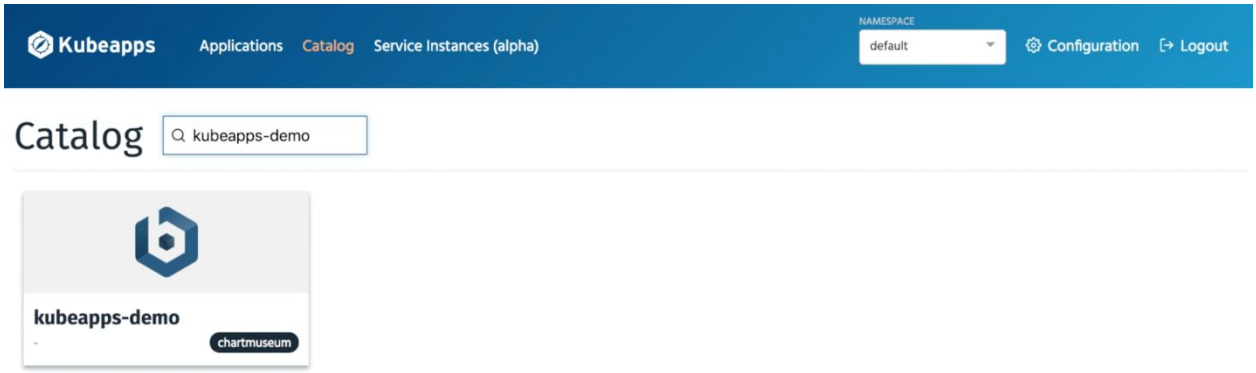
How To Deploy An Application On A Cluster Using Kubeapps

The instructions below will walk you through the process of deploying a sample application on your Kubernetes cluster using the Kubeapps UI. For the sake of simplicity, a Java Tomcat application with MySQL, that has previously been packaged with Bitnami Stacksmith, will be used, but the process is the same if you want to deploy an application from your own catalog.

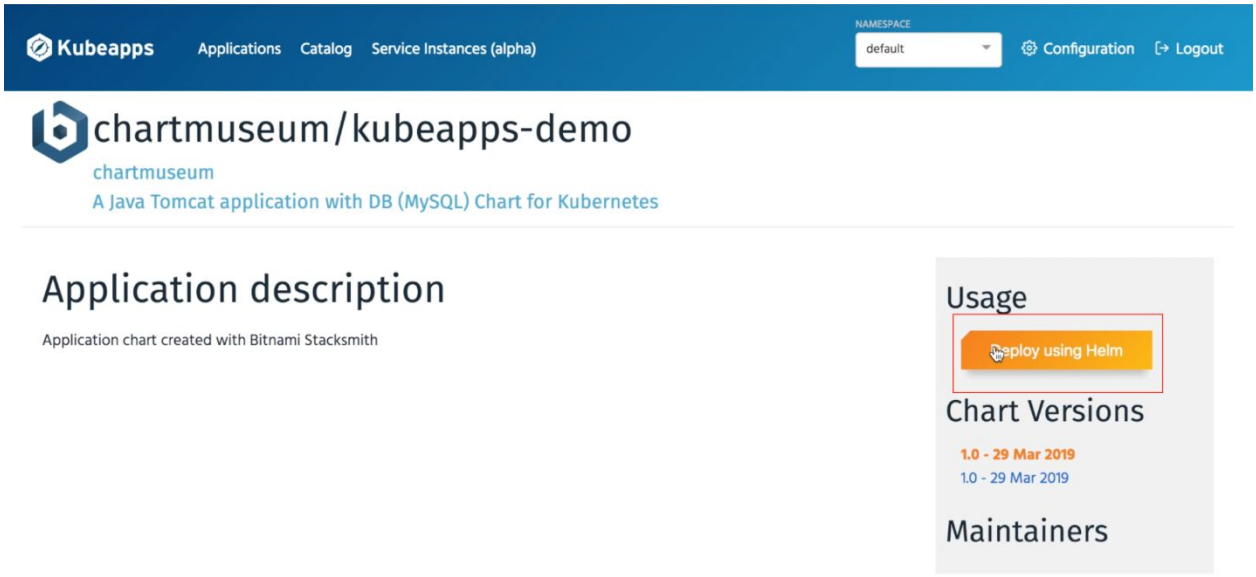
Deploy A Java Tomcat Application With MySQL

Once the Kubeapps Dashboard is up and running, you can start deploying applications into your cluster.

- Use the "Catalog" page in the dashboard to search for the application from the list of charts in any of the configured Helm chart repositories.

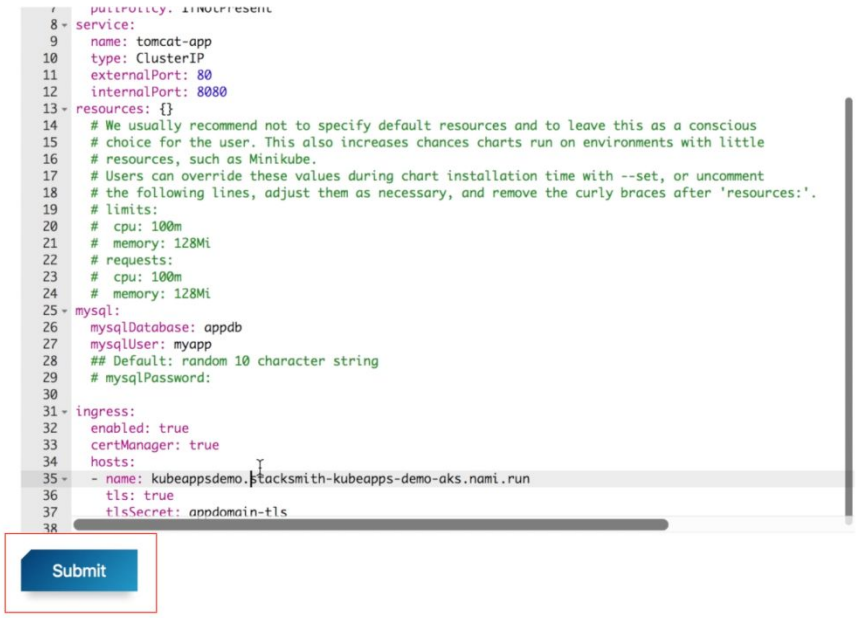


- Click on "Deploy using Helm."

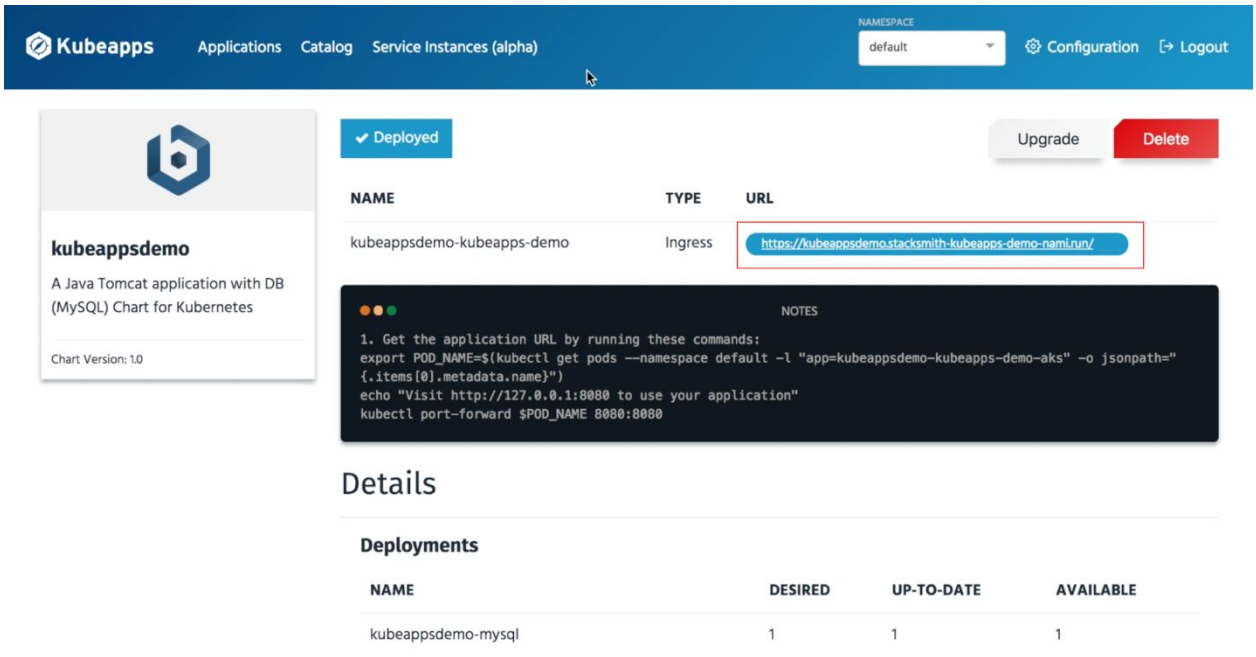


- You will be prompted for the release name and values for the application.

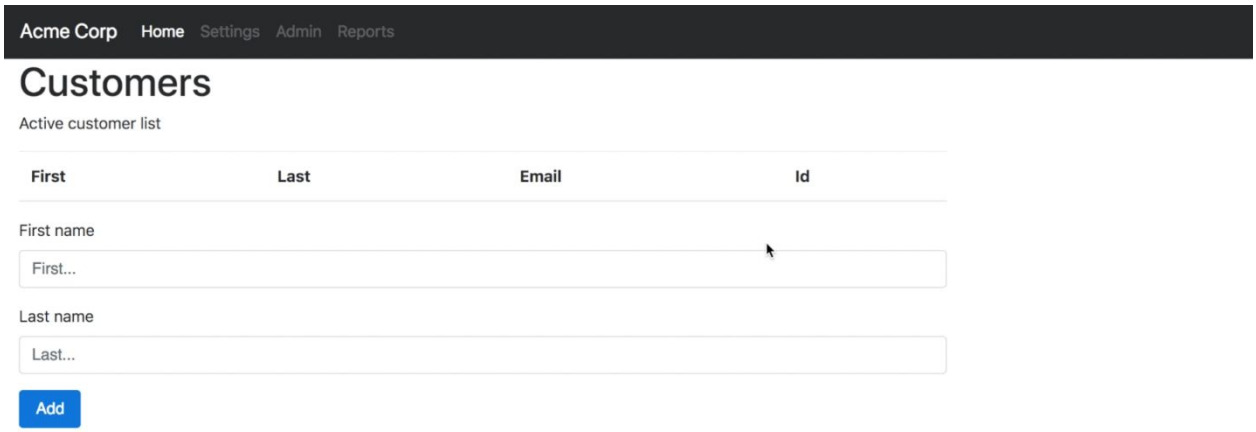
- Click on "Submit." The application will be deployed. You will be able to track the new Kubernetes deployment directly from the browser.



Use the URLs shown to directly access the application. Note that, depending on your cloud provider of choice, it may take some time for an access URL to be available for the application and the service will stay in a "Pending" state until a URL is assigned. If using Minikube, run `minikube tunnel` in your terminal in order for an IP address to be assigned to your application.



Congrats! Now you are able to access your application from your browser:



Kubeapps, The Bitnami Open-Source Project That Helps You Provide And Manage Kubernetes Deployments From A Single UI

As you have seen in this blog post, Kubeapps not only allows you to deploy and manage applications in Kubernetes clusters from a visual dashboard, but it also provides a single place for your organization where your users can discover new applications and IT services that are compliant with your internal requirements and best practices. Thus, it allows users to deploy only the approved IT services they need.

Start using Kubeapps now to provide your team with charts that are compliant with your company standards!

> Bitnami provides supporting Open Source tools for Kubernetes users such as: Bitnami Kubernetes Production Runtime (BKPR), Kubeless, Kubecfg, Kubewatch, or Sealed Secrets. And last but not least, the Bitnami key project for Kubernetes: Kubeapps.

Upgrade and Delete Applications From the Kubeapps UI

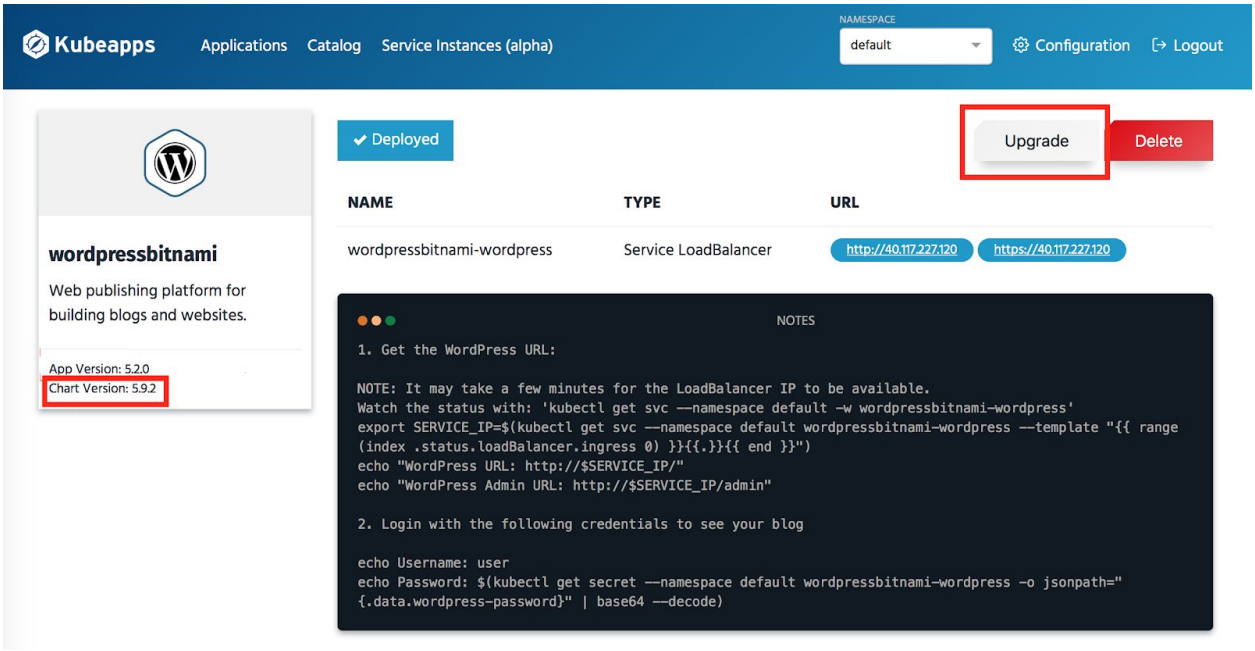
One of the Kubeapps features that cluster operators will probably appreciate is the possibility of upgrading and removing applications directly from the Kubeapps UI. That way, they can manage their clusters without the need of typing a series of commands in a terminal. Managing deployments on Kubernetes never was so easy!

Upgrade Existing Application Deployments

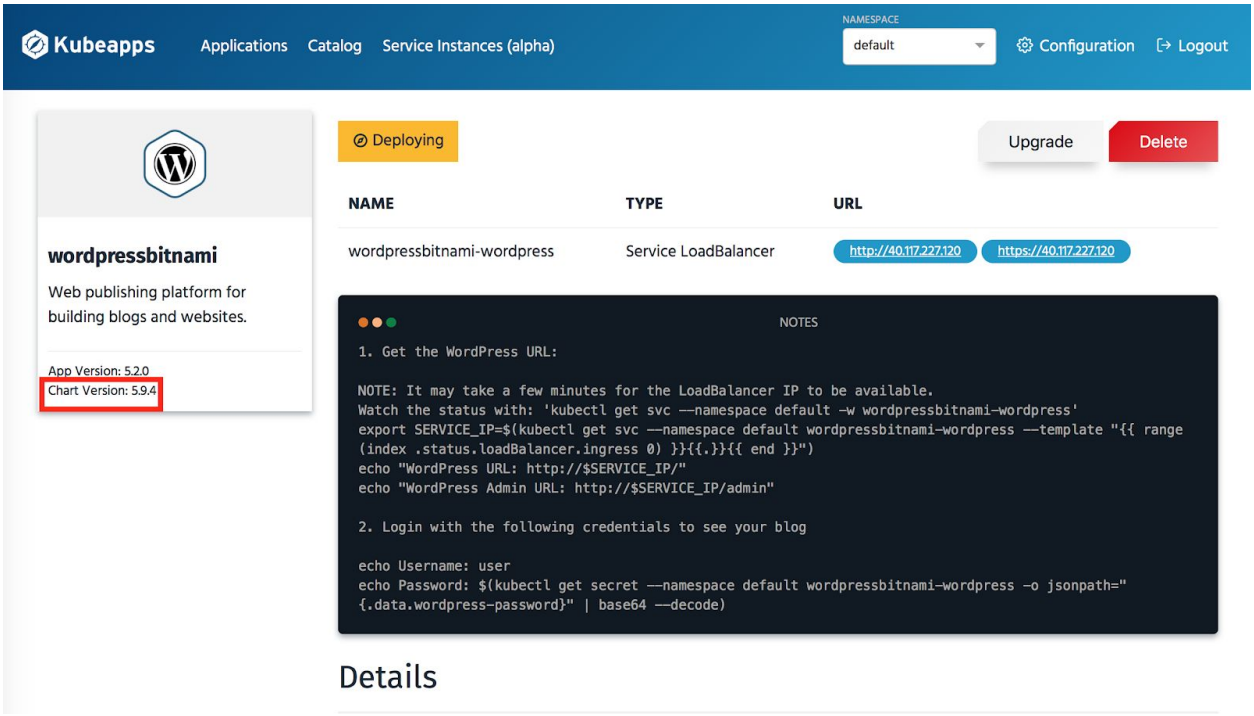
In case that a new version of an application is released or if you have introduced changes in your code and added that new version to your catalog, upgrading your application deployment is as simple as a one-single click.

To upgrade an application, follow the instructions below:

- On the "Applications" page, select the deployment you want to upgrade.
- Click the "Upgrade" button.



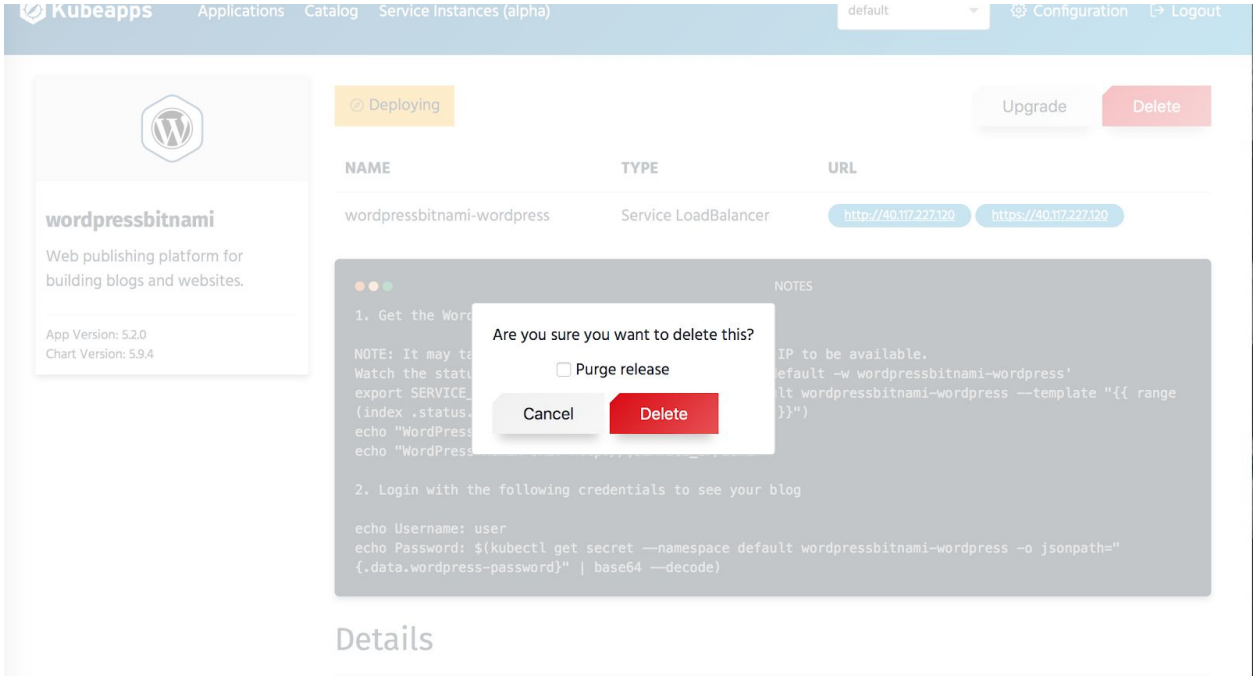
- Select the repository in which you want to deploy the new version of the selected application.
- Select the version of the application you want to deploy and make the corresponding changes in the values.yaml file if necessary. Click “Submit”.
- A new deployment of the application will start as shown below:



That’s all! Your application will be automatically upgraded!

Remove Existing Application Deployments

You can remove any of the applications from your cluster by clicking the "Delete" button on the application's status page.



Are you interested in using Kubeapps on the VMware Enterprise PKS? Read this blog post to learn how to install it on your cluster via Harbor.

How Kubeapps Enforces the Security of Your Kubernetes Deployments

Exploring the Security of Helm

Authored by Angus Lees, Kubernetes developer

December 12, 2017

Bitnami has been a part of the Helm community for a long while, but I personally started looking at Helm only a few weeks ago in the context of our work on Kubeapps - a package agnostic launchpad for Kubernetes applications. I was writing the manifests to deploy all the necessary tooling and started digging into a secure deployment configuration for Helm.
The executive summary is that I felt that Helm/Tiller needed to be properly configured to be operated securely in a shared/production environment.

This post is a breakdown of some of what I've learned, and some of the approaches available to secure tiller within a cluster. The wider discussion about these issues has led to a good summary doc from the Helm folks - in the following, I'm going to expand on many of the details. I have tried to include plenty of worked examples so readers can see how to explore their own Tiller setup before and after making any changes.

Breaking Down The Problem

There are several angles from which someone might try to abuse Helm/Tiller:

- **A privileged API user,** such as a cluster-admin. We actually want these users to have access to the full power and convenience of helm charts.
- **A low-privilege API user,** such as a user who has been restricted to a single namespace using RBAC. We would like to allow these users to install charts into their namespace, but not affect other namespaces.
- **An in-cluster process,** such as a compromised webserver. There is no reason these processes should install helm charts, and we want to prevent them from doing so.
- **A hostile chart author** can create a chart containing unexpected resources. These can either escalate one of the other groups above, or run other malicious jobs.

It is easy to mash all the issues and conversations together, but I recommend keeping the above separation in mind when considering your own needs and solutions.
In the rest of this blog, I will show you what each of these risks actually means and show you a remediation path.

Helm Architecture

First, some required background. "Helm" is a project and a command line tool to manage bundles of Kubernetes manifests called charts. helm (the command line tool) works by talking to a server-side component that runs in your cluster called Tiller. Tiller by default runs as the tiller-deploy Deployment in kube-system namespace. Much of the power and flexibility of Helm comes from the fact that charts can contain just about any Kubernetes resource. There are good docs on the helm website if you want to read about any of this in more depth.
The helm -> tiller communication happens over gRPC connections. These are not your usual HTTP, but you can think of it in a similar way: `helm` sends a command and arguments down the gRPC TCP channel, tiller does the work, and then returns a result which helm usually displays to the user in some form.
One of the first concerning aspects of Tiller is that it is able to create just about any Kubernetes API resource in a cluster. It has to, because that is its job when installing a chart! Tiller performs no authentication by default - so if we can talk to Tiller, then we can tell it to install just about anything.

In-Cluster Attacks

This is one of the most alarming of the cases above, so we will focus on it first. The design of Kubernetes RBAC, namespaces, pods, etc strives to isolate serving jobs from each other. If a webserver is compromised (for example), you really want that rogue process to stay contained and not be able to easily escalate and exploit the rest of your cluster.

By default Tiller exposes its gRPC port inside the cluster, without authentication. That means **any** pod inside your cluster can ask tiller to install a chart that installs new ClusterRoles granting the local pod arbitrary, elevated RBAC privileges: Game over, thanks for playing.

What Are We Talking About, Exactly?

Here is a **complete worked example,** so you can see what's involved, and how to test your own cluster.
First, install tiller. We're going to install tiller using the defaults.

```
$ helm init
$HELM_HOME has been configured at /home/gus/.helm.

Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.
Happy Helming!
```

What does this do? `helm init` creates a Service and a Deployment, both called `tiller-deploy` in the `kube-system` namespace. Note the default tiller port is 44134 - we're going to use that later.

```
$ kubectl -n kube-system get svc/tiller-deploy deploy/tiller-deploy
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
svc/tiller-deploy   ClusterIP   10.0.0.216   <none>        44134/TCP
1h

NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
deploy/tiller-deploy   1         1         1            1
1h
```

A ClusterIP Service creates a hostname and port that can be accessed from elsewhere inside the cluster. **What happens if we poke at those directly?**

For now, let's test our access to tiller from inside the cluster. The following sets up a basic shell environment running in a temporary pod in the "default" namespace. Importantly, this pod has no special privileges.

```
$ kubectl run -n default --quiet --rm --restart=Never -ti
--image=anguslees/helm-security-post incluster
root@incluster:/# helm version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Error: cannot connect to Tiller
```

Good! `"Error: cannot connect to Tiller"` is exactly what we want to see! But wait, we saw earlier that tiller installs a service by default. **What happens if we try to talk to that host/port directly?**

```
root@incluster:/# telnet tiller-deploy.kube-system 44134
Trying 10.0.0.216...
Connected to tiller-deploy.kube-system.svc.cluster.local.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
```

Huh, that looks like I can connect to Tiller. Sure enough, the earlier "cannot connect" error wasn't quite true and what actually failed was the `helm` CLI trying to discover Tiller. If we bypass the discovery step, **the actual connection to tiller works just fine:**

```
root@incluster:/# helm --host tiller-deploy.kube-system:44134 version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0",
GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4",
GitTreeState:"clean"}
```

Just to drive home exactly what this means, here's the rest of an example exploit. I created a very simple chart that just binds the "default" service account to a wildcard "allow everything" RBAC ClusterRole. You can see that installing this chart immediately allows my previously-unprivileged pod to read the secrets from `kube-system` (for example).

```
root@incluster:/# kubectl get secrets -n kube-system
Error from server (Forbidden): secrets is forbidden: User
"system:serviceaccount:default:default" cannot list secrets in the
namespace "kube-system"

root@incluster:/# helm --host tiller-deploy.kube-system:44134 install
/pwnchart
NAME:   dozing-moose
LAST DEPLOYED: Fri Dec  1 11:24:22 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1beta1/ClusterRole
NAME           AGE
all-your-base  0s

==> v1beta1/ClusterRoleBinding
NAME          AGE
belong-to-us  0s


root@incluster:/# kubectl get secrets -n kube-system
NAME                                  TYPE
DATA      AGE
attachdetach-controller-token-v2mbl
kubernetes.io/service-account-token   3         1d
bootstrap-signer-token-svl22
kubernetes.io/service-account-token   3         1d
<etc ...>
```

How Can We Close Down The Tiller Port?

Luckily this is easy for most users. If you only use Tiller through the `helm` CLI tool, this is sufficient to secure a default tiller install from attacks inside your cluster:

```
kubectl -n kube-system delete service tiller-deploy
kubectl -n kube-system patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'
```

What does this do? This patches the Tiller Deployment to run with `--listen=localhost:44134` flag. This flag causes tiller to listen for gRPC connections only on the localhost address inside the pod. The Service and ports declaration are now useless, so we remove them too just to be nice.

Note that just removing the Service is not sufficient. The Tiller port is still exposed, if you know the pod address.

```
$ kubectl -n kube-system delete service tiller-deploy
service "tiller-deploy" deleted
$ kubectl get pods -n kube-system -l app=helm,name=tiller -o
custom-columns=NAME:.metadata.name,PODIP:.status.podIP
NAME                             PODIP
tiller-deploy-84b97f465c-pdfdp   172.17.0.3
$ kubectl run -n default --quiet --rm --restart=Never -ti
--image=anguslees/helm-security-post incluster
root@incluster:/# helm --host=tiller-deploy.kube-system:44134 version
```

```
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Error: cannot connect to Tiller
root@incluster:/# helm --host=172.17.0.3:44134 version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0",
GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4",
GitTreeState:"clean"}
```

The last command above demonstrates that tiller is still exposed on the pod IP address. Continuing this example to show that restarting Tiller with `--listen=localhost:44134` does restrict access:

```
$ kubectl -n kube-system patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'

deployment "tiller-deploy" patched
$ kubectl run -n default --quiet --rm --restart=Never -ti
--image=anguslees/helm-security-post incluster
root@incluster:/# helm --host=172.17.0.3:44134 version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Error: cannot connect to Tiller
```

I have never been happier to see a "cannot connect" error message. **By listening on localhost only, tiller is no longer accessible within the cluster.**

Note that the `helm` CLI tool still works, when used by privileged users. This is because the CLI tool creates a temporary port forward to the tiller-deploy pod directly, and then communicates down the forwarded connection. The port-forwarded connection "pops out" in the pod's own network namespace, and so can connect to the pod's idea of localhost just fine. I'll come back to the `helm` CLI again later.

```
$ kubectl auth can-i create pods --subresource portforward -n
kube-system
yes
$ helm version
Client: &version.Version{SemVer:"v2.7.0",
GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0",
GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4",
GitTreeState:"clean"}
```

What If I *Need* To Access Tiller Port In-Cluster?

There are some 3rd-party services that build on Tiller, such as monocular and landscaper. These fall into a few categories depending on how they work.
Some tools use `helm` CLI directly (eg: a DIY jenkins CI/CD pipeline). These will work just fine in-cluster provided the service account they are running as is able to create a port-forward to the tiller pod. See the section on low-privileged API users below.

Some tools talk to the gRPC port directly. You have two basic choices with these: you can either set up Tiller's gRPC TLS authentication, or bundle all the related services into the same pod as Tiller.

Tiller With TLS-Authenticated GRPC

There are some good upstream docs for this, however the TLS configuration was **ineffective until v2.7.2.**
For simplicity in the following example, I've wrapped up all the TLS certificate generation in a simple script that uses `openssl` - any other method of generating these keys is fine.

```
$ : Ensure at least v2.7.2 !
$ helm version --client --short
Client: v2.7.2+g8478fb4
$ ./tls.make
(output skipped)
$ kubectl delete deploy -n kube-system tiller-deploy
deployment "tiller-deploy" deleted
$ helm init --tiller-tls --tiller-tls-cert ./tiller.crt
--tiller-tls-key ./tiller.key --tiller-tls-verify --tls-ca-cert
ca.crt
$HELM_HOME has been configured at /home/gus/.helm.
(Use --client-only to suppress this message, or --upgrade to upgrade
Tiller to the current version.)
Happy Helming!
```

What has this done? This deletes the old tiller Deployment (to clean up after our earlier `--listen=localhost:44134` change), and then installs a fresh tiller with the **gRPC port open inside the cluster, but *secured with TLS certificates.** If you are curious, the tiller TLS files have been uploaded into the `tiller-secret` Secret in `kube-system` namespace.
Now we can retry our earlier in-cluster test:

```
$ kubectl run -n default --quiet --rm --restart=Never -ti
--image=anguslees/helm-security-post incluster
root@incluster:/# telnet tiller-deploy.kube-system 44134
Trying 10.0.0.6...
Connected to tiller-deploy.kube-system.svc.cluster.local.
Escape character is '^]'.
```

```
^]
telnet> quit
Connection closed.
root@incluster:/# helm --host=tiller-deploy.kube-system:44134 version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Error: cannot connect to Tiller
```

Note the port is open (`telnet` can connect to it), but the Tiller server immediately closes the gRPC connection because we don't have a suitable TLS certificate. Good.

The same is true even when the client tries to connect from localhost over the port-forward. **With TLS enabled, everyone has to use TLS certs.**

```
$ helm version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Error: cannot connect to Tiller
$ ./tls.make myclient.crt myclient.key
(output skipped)
$ helm --tls --tls-ca-cert ca.crt --tls-cert myclient.crt --tls-key
myclient.key version
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
$ cp ca.crt $(helm home)/ca.pem
$ cp myclient.crt $(helm home)/cert.pem
$ cp myclient.key $(helm home)/key.pem
$ : The following looks like a bug!
$ helm version --tls
could not read x509 key pair (cert: "/cert.pem", key: "/key.pem"):
can't load key pair from cert /cert.pem and key /key.pem: open
```

```
/cert.pem: no such file or directory
$ export HELM_HOME=$(helm home)
$ helm version --tls
Client: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2",
GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6",
GitTreeState:"clean"}
```

Note that we now have to present a valid TLS certificate, even when using the port-forward method. We can save some typing by copying the TLS files into the helm config dir, but note there appears to be a bug that requires `HELM_HOME` to be explicitly set, and we still need to use `helm --tls` everywhere. This is annoying, but is **necessary for setups where tiller needs to be exposed** to 3rd party services within the cluster.

Alternative: Sidecars

Some tiller clients (eg monocular) do not support TLS-secured gRPC connections yet. The approach we took with kubeapps dashboard was to move the `monocular-api` component and tiller into the same pod, and use the earlier `--listen` flag to restrict tiller to localhost within that pod.

The details are messy but straightforward. You can see the relevant portion of the kubeapps manifests if you are curious. Note these configs are written using `kubecfg` (from the *ksonnet* project).

Limited Access For Low-Privileged Users

The above is great, but completely ignores authorization. Even with TLS configured, tiller is still all-or-nothing: If you can send commands, then, you can install anything you want and effectively own the cluster.

What if you want to allow your users to use helm, but still restrict what a chart can do?

Tiller Per Namespace

The basic approach is to use RBAC to limit the tiller service account. We can then have **many tiller instances,** each running with different service account, and different RBAC restrictions. By controlling who is allowed to access which tiller instance, we can control what those users are allowed to do via tiller.

One obvious example is providing a tiller instance per namespace, limited to only creating resources in that namespace.

```
$ NAMESPACE=default
$ kubectl -n $NAMESPACE create serviceaccount tiller
serviceaccount "tiller" created
$ kubectl -n $NAMESPACE create role tiller --verb '*' --resource
'services,deployments,configmaps,secrets,persistentvolumeclaims'
role "tiller" created
$ kubectl -n $NAMESPACE create rolebinding tiller --role tiller
--serviceaccount ${NAMESPACE}:tiller
rolebinding "tiller" created
$ kubectl create clusterrole tiller --verb get --resource namespaces
clusterrole "tiller" created
$ kubectl create clusterrolebinding tiller --clusterrole tiller
--serviceaccount ${NAMESPACE}:tiller
clusterrolebinding "tiller" created
$ helm init --service-account=tiller --tiller-namespace=$NAMESPACE
$HELM_HOME has been configured at /home/gus/.helm.

Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.
Happy Helming!
$ kubectl -n $NAMESPACE delete service tiller-deploy
service "tiller-deploy" deleted
$ kubectl -n $NAMESPACE patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'
deployment "tiller-deploy" patched
```

This creates a new service account, with the RBAC permissions to do anything to services, deployments, configmaps, secrets and PVCs (extend as desired) within the specfied

namespace. We then install a new tiller instance in this namespace using the new service account, and apply our earlier patches to restrict the tiller port to localhost.

Let's test it out:

```
$ helm --tiller-namespace=$NAMESPACE install stable/wordpress
NAME:   intent-mite
LAST DEPLOYED: Sun Dec  3 13:28:56 2017
NAMESPACE: default
STATUS: DEPLOYED
(snip)
```

```
$ helm --tiller-namespace=$NAMESPACE install ./pwnchart
Error: release boisterous-quetzal failed:
clusterroles.rbac.authorization.k8s.io is forbidden: User
"system:serviceaccount:default:tiller" cannot create
clusterroles.rbac.authorization.k8s.io at the cluster scope
```

Success! A "nice" single-namespace chart like wordpress installed successfully, but we were **unable to install a chart that tried to modify a global resource.**

Note the above setup needs to be repeated for every namespace (or desired tiller isolation boundary). Each `helm` CLI user will also need to be able to find and port-forward to "their" tiller instance. In the RBAC rule language, this means the ability to "list" "pods" and "create" a "pods/portforward" resource.

The downside of this approach is having to manage the setup explicitly for each namespace, and the cumulative runtime overhead of all those separate tiller instances. In our Bitnami internal development cluster (for example), we have a namespace for each developer and this approach translates to almost an entire VM dedicated to just running tillers.

Helm CRD

With Custom Resource Definitions (CRDs) becoming a standard extension mechanism for Kubernetes, I wanted to see what Helm would look like as a conventional Kubernetes addon. So I wrote a simple Helm CRD controller. This allows you to **manage Helm charts by creating regular Kubernetes resources** instead of using the `helm` CLI and sending gRPC commands.

Using Kubernetes resources has a **number of benefits:**

- The tiller gRPC channel is not exposed
- HelmRelease resource can be restricted using RBAC as usual
- Integrates with other YAML-based tools and declarative workflows, like `kubectl` and kubecfg
- No need for the `helm` CLI for simple operations

The helm-crd controller also comes with an optional `kubectl` plugin to simplify command-line use, but it can also be operated directly with regular `kubectl` and YAML files. Using the plugin looks like this:

```
$ mkdir -p ~/.kube/plugins/helm
$ pushd ~/.kube/plugins/helm
$ wget \

https://raw.githubusercontent.com/bitnami/helm-crd/master/plugin/helm
/helm \

https://raw.githubusercontent.com/bitnami/helm-crd/master/plugin/helm
/plugin.yaml
$ chmod +x helm
$ popd
$ kubectl delete deploy -n kube-system tiller-deploy
deployment "tiller-deploy" deleted
$ kubectl plugin helm init
customresourcedefinition "helmreleases.helm.bitnami.com" created
deployment "tiller-deploy" created
```

This installs the `kubectl` plugin, and then uses the plugin to install tiller, the helm-crd controller, and `HelmRelease` custom resource definition. Note that helm-crd currently always installs Tiller into `kube-system`, with the CRD controller and tiller running in the same pod and the Tiller gRPC port restricted to localhost (as described earlier).

```
$ kubectl plugin helm install mariadb --version 2.0.1
helmrelease "mariadb-5w9hd" created
$ kubectl get helmrelease mariadb-5w9hd -o yaml
```

```
apiVersion: helm.bitnami.com/v1
kind: HelmRelease
metadata:
  clusterName: ""
  creationTimestamp: 2017-12-03T03:13:40Z
  deletionGracePeriodSeconds: null
  deletionTimestamp: null
  generateName: mariadb-
  name: mariadb-5w9hd
  namespace: default
  resourceVersion: "209450"
  selfLink:
/apis/helm.bitnami.com/v1/namespaces/default/helmreleases/mariadb-5w9
hd
  uid: f647e29f-d7d7-11e7-be67-525400793f03
spec:
  chartName: mariadb
  repoUrl: https://kubernetes-charts.storage.googleapis.com
  values: ""
  version: 2.0.1
```

The controller has only seen light testing so far, but is simple, and is usable right now. Currently it does not deal with the "hostile chart" problem, but I have a new tiller feature that will allow me to create a separate "cluster level" CRD resource, much like the Role/ClusterRole split in RBAC.

Additional usage and feedback is valuable, and we hope to be able to use this experience to inform the "Helm v3" discussions starting in February.

Conclusion

This was a long doc. The basic take-away is that when using helm in a shared/production environment, the `helm init` default installation is not sufficient.

- If you have *no 3rd-party services* that talk to tiller: Restart tiller with `--listen=localhost:44134` flag.
- If you have *3rd-party services* that talk to tiller: Configure TLS authentication.
- If you need to offer *different levels of access* via tiller: Complicated. Multiple tiller instances, or Helm-crd.

NOTE: Enforce the security of your cluster by assigning different roles to users to determine which operations are allowed to perform. Learn how to configure service accounts and user roles using Kubeapps.

Running Helm in Production: Security Best Practices

Authored by Andres Martinez, Software Engineer

February 25, 2019

Helm has become one the most popular package managers for Kubernetes. The goal of Helm is to help you manage Kubernetes applications using Charts. Helm charts are just "packages" that you can directly install in your Kubernetes cluster. They are really useful since they abstract all the complexity around ConfigMaps, Deployments, Volumes, etc. that otherwise you need to handle one by one, to deploy applications in Kubernetes.

When using Helm in production (i.e. in a Kubernetes cluster with security policies), it's necessary to understand how to properly set up this tool to avoid security issues. In this post, I will walk through some of the security challenges produced by Helm, explaining security best practices to avoid these issues and how Bitnami can help you overcome them with Kubeapps: An open source, web-based UI for deploying and managing applications in Kubernetes.

Installing Helm

Helm is very easy to install: once you get the `helm` CLI, you just execute a single command:

```
$ helm init
$HELM_HOME has been configured at /home/andres/.helm.

Tiller (the Helm server-side component) has been installed in your
Kubernetes cluster.

Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation, see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
```

So now you are ready to install your favorite applications in Kubernetes! But wait, something else should catch your attention:
Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.

That's indeed disturbing. Let's dig into that a little bit more.

Helm Security Challenges

Angus Lees, a Bitnami Kubernetes developer, wrote a very detailed article about what the security challenges of Helm are. The main issue is that Helm requires a server side named Tiller. This component is in charge of contacting the Kubernetes API in order to install, on behalf of the user, anything specified in a Helm chart.

This implies that Tiller:

- **will usually need admin privileges:** If a user wants to install a chart that contains any cluster-wide element like a ClusterRole or a CustomResourceDefinition (CRD), Tiller should be privileged enough to create or delete those resources.
- **should be accessible to any authenticated user:** Any valid user of the cluster may require access to install a chart.

That leads to a now-obvious security issue: escalation of privileges. Suddenly, users with minimum privileges are able to interact with the cluster as if they were administrators. The problem is bigger if a Kubernetes pod gets compromised: that compromised pod is also able to access the cluster as an administrator. That's indeed disturbing.

Suddenly, users with minimum privileges are able to interact with the cluster as if they were administrators. The problem is bigger if a Kubernetes pod gets compromised: that compromised pod is also able to access the cluster as an administrator.

Mitigating The Issues

The official Helm documentation explains a few hints to mitigate these problems. Unfortunately, they don't directly suit this case:

- **Reducing Tiller permissions** may be an obvious option. That doesn't fit our needs though. You'll probably want cluster administrators to be able to install charts with cluster-wide components.
- **Securing Tiller endpoint with a TLS certificate.** With this certificate, users not only need to have a valid user, but they also need access to Tiller certificate to contact it. A compromised pod isn't able to access Tiller since you should restrict access to the certificate by default. The problem is that managing access to the certificate is difficult to maintain. Now cluster administrators need to apply rules to allow or deny access for every new user.
- **Running a Tiller instance per namespace.** This way you can reduce Tiller permissions for certain instances, while leaving others privileged. Again, the downside with this solution is that it's difficult to maintain and now you are wasting resources, having duplicated deployments.

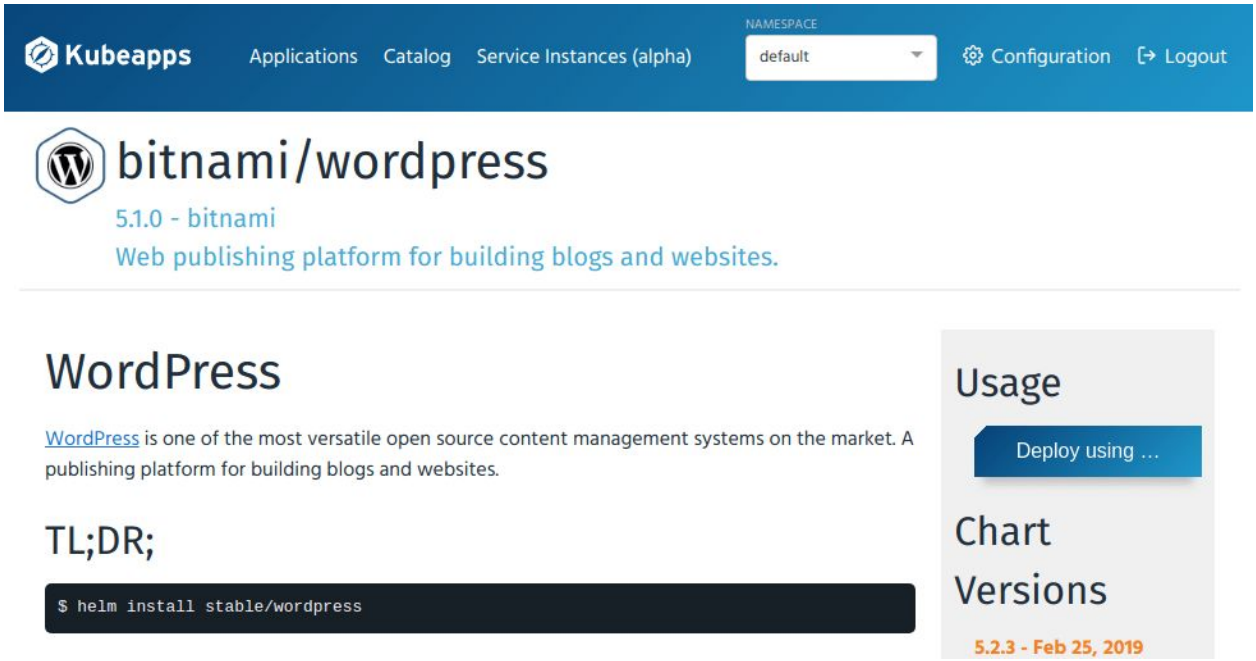
Tillerless And Helm V3

Helm maintainers are aware of the vulnerabilities that having a server side for Helm causes, so in the next major version, they'll get rid of Tiller. If you are curious about what's going to happen with this new version, find the design proposal here. Unfortunately, an official release date is not yet available, so for the time being, it is necessary to stick with the current version 2.
A special mention is deserved for what's called Tillerless Helm, which is about **running Tiller in your local host** rather than in the Kubernetes cluster. Tiller momentarily runs using the authentication information of the user who executes it, so it's not possible to escalate privileges.

Using this approach has a downside though. You still need to store information about the charts that you install in the cluster. That means that anyone using this solution will need to configure the namespace they are allowed to use, and they will need to run several commands (in different terminals) to deploy a chart. There is a plugin that does this for you, but in any case, this **will divert from the default experience.**

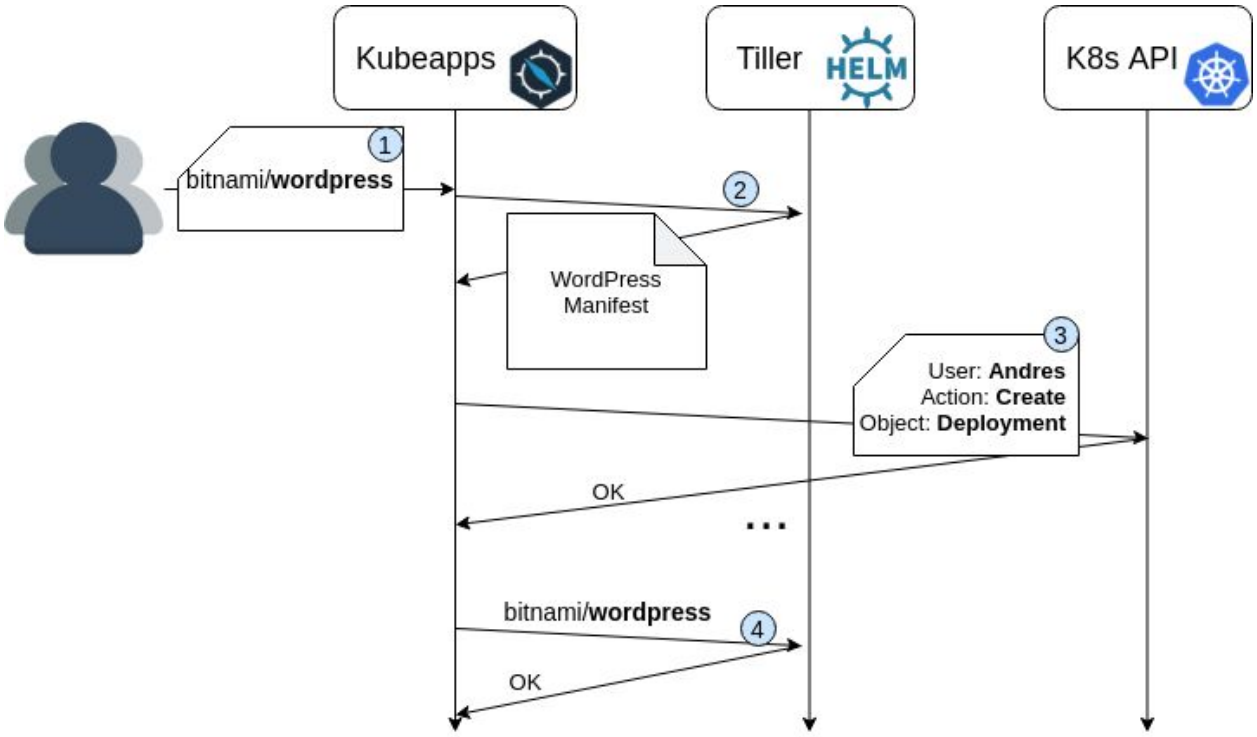
Using Kubeapps And Tiller-Proxy

Kubeapps is a web-based UI for managing applications in Kubernetes clusters. In other words, it allows you to discover and install Helm charts without the `helm` CLI, using a web interface.



Kubernetes Role-Based Access Control (RBAC) system backs Kubeapps. This means that to sign in, you need a Kubernetes API token, which is really easy to obtain. Learn how to do so in the getting started guide. Once users are logged in, they will be authenticated as specific Kubernetes users and they won't be able to escalate privileges. To achieve this, we have developed an authorization proxy that validates any action targeting Tiller. This simplified diagram explains how we do it:

Find out how to package and launch trusted applications on Kubernetes combining Stacksmith and Kubeapps.



1. The user requests to install a chart (i.e. bitnami/wordpress).
2. Before installing the chart, Kubeapps resolves the manifest with Tiller. This manifest contains all the resources required for the chart.
3. For each one of the resources, Kubeapps checks that the authenticated user has permissions to create it in the given namespace. To do so, Kubeapps uses the Authorization API and is able to differ if the user has permissions to perform the requested action (i.e. create a Deployment in the namespace "default").
4. Kubeapps installs the chart only if the request is valid.

With that set-up, it's really easy for users to install Helm charts without the security disadvantages of using a single Tiller for a Kubernetes cluster.

If you are interested in learning how to set up Helm, Tiller, and Kubeapps securely, check out our step-by-step guide.

Happy Helming!

Integrating Kubeapps with the Kubernetes Service Catalog

Kubernetes Service Catalog Kubeapps Integration

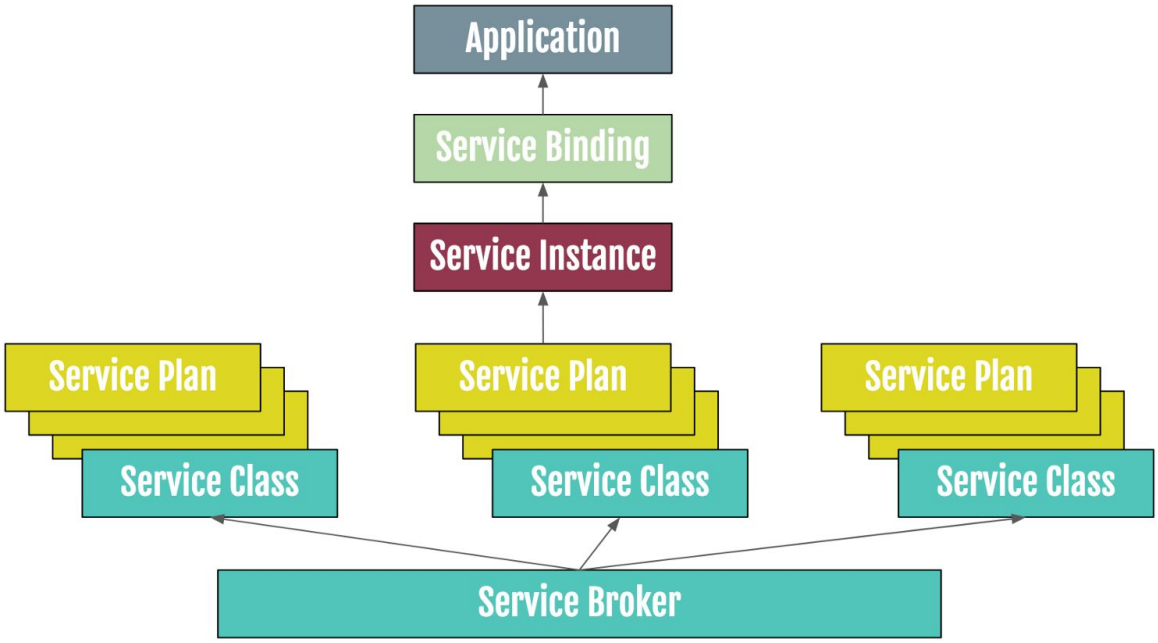
Kubernetes is a great platform to run containerized workloads in production. Developers can express their applications orchestration in a declarative way, and set up their CI/CD workflows to deploy into production continuously.

However, many of the cloud native applications depend on managed services offered by the clouds. Developers use databases, pubsub queues, storage, etc. from the major clouds knowing that those services already implement many of the features they would need to manage themselves otherwise (backups, HA, etc.). Combining Kubernetes with external services available from public cloud providers can be a powerful way to deploy cloud native applications. **Developers can focus on deploying their applications to their Kubernetes clusters,** while delegating things like database management to a public cloud managed service.

In the past, this integration needed to happen outside Kubernetes. Developers needed to provision those services, then connect them to their applications creating secrets with the right credentials, breaking their declarative GitOps workflows. The Service Catalog bridges these two worlds by allowing developers instantiate services outside Kubernetes directly from their Kubernetes cluster, in a full declarative way.

> Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider.

Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider. These services are provided by a Service Broker, which is an endpoint talking to these providers. Once the cluster administrator deploys a ClusterServiceBroker, several ClusterServiceClasses and ClusterServicePlans will be available in the cluster for users to provision those services. To provision a service, users will create a ServiceInstance object and to connect it to their application they will create a ServiceBinding object.

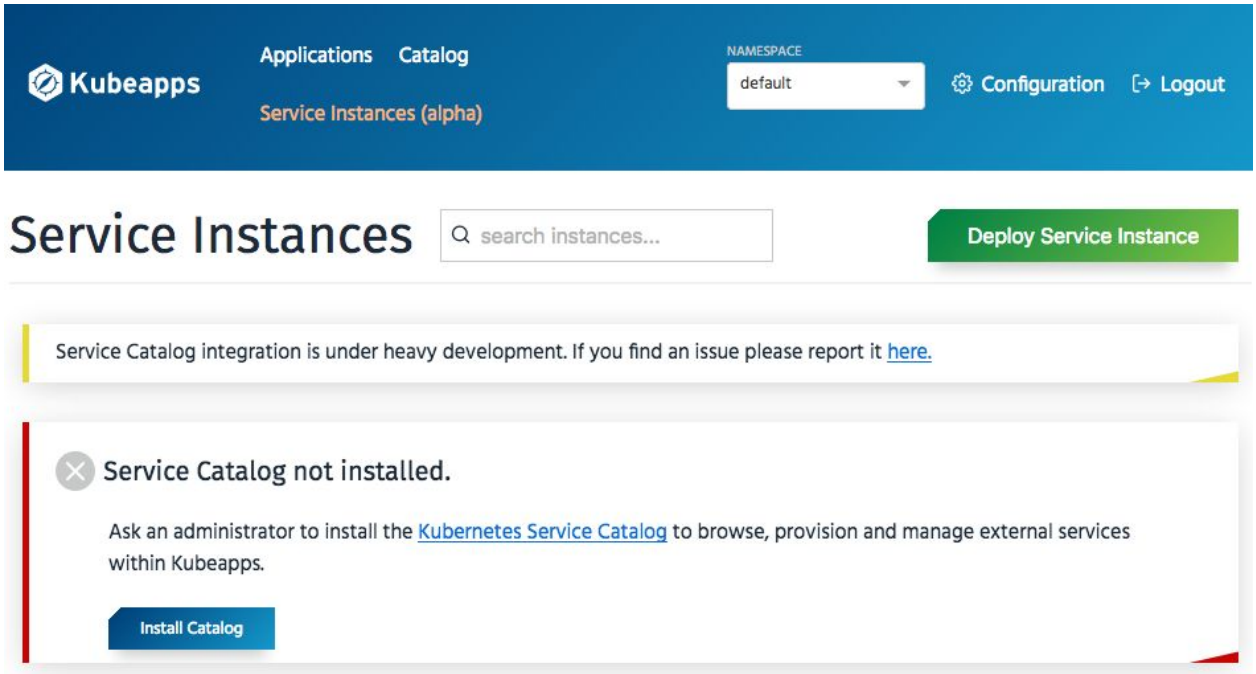


Service Catalog and Kubeapps

Kubeapps has native integration with the Service Catalog and allows Kubeapps users to provision external cloud services directly from the Kubeapps interface.
In this tutorial we will explain how to deploy the Service Catalog into your cluster, we will configure two Service Brokers (GCP and Azure) and we will provision some cloud services that we will then use in other applications.
This tutorial assumes that you already have a Kubernetes cluster setup with Helm and Kubeapps. If you don't have Helm or Kubeapps you can follow the installation instructions for Kubeapps.

Deploy Service Catalog

The Service Catalog is distributed as a Helm Chart and it is ready to be deployed with Kubeapps.
To deploy it in your cluster, navigate to “Service Instances” and click on “Install Catalog.”



You will deploy the Service Catalog as any other Helm Chart installed through Kubeapps. We recommend to at least change the following value in *values.yaml:* `asyncBindingOperationsEnabled: true`

This value is needed for some of the GCP Service Classes to work properly. Alternatively, you can deploy the Service Catalog using the Helm CLI:

```
helm repo add svc-cat
https://svc-catalog-charts.storage.googleapis.com
helm repo update
helm install svc-cat/catalog --name catalog --namespace catalog --set
asyncBindingOperationsEnabled=true
```

This will deploy the Service Catalog API server and its controller into a namespace called `catalog`.

Deploy the Azure Service Broker

To deploy the Open Service Broker for Azure the first thing that you will need to do is to configure your Azure account properly. The services you are going to use are paid-for and, therefore, you need to have your account and billing settings set up properly.
Once you have your account set up, follow the instructions in the Open Service Broker for Azure on AKS Quickstart. Although the Quickstart mentions AKS, the instructions should work in any Kubernetes cluster.
To check that the broker has been successfully deployed run the following:

```
kubectl get ClusterServiceBroker osba
```

If the Broker has been successfully installed and the catalog has been properly downloaded you should get the following output:

```
NAME         URL
STATUS   AGE
osba
https://osba-open-service-broker-azure.osba.svc.cluster.local
Ready    6m
```

Deploy the GCP Service Broker

To deploy the GCP Service Broker you will need to configure a GCP account and project. The services you are going to use are paid-for and, therefore, you need to have your account and billing settings set up properly.
Once you have your account set up, follow the instructions at https://cloud.google.com/kubernetes-engine/docs/how-to/add-on/service-catalog/install-service-catalog. You can skip the part of deploying the Service Catalog (which we already deployed using the Helm Chart). Although the instructions mention GKE, they should work in any Kubernetes cluster.

To check that the broker has been successfully deployed run the following:

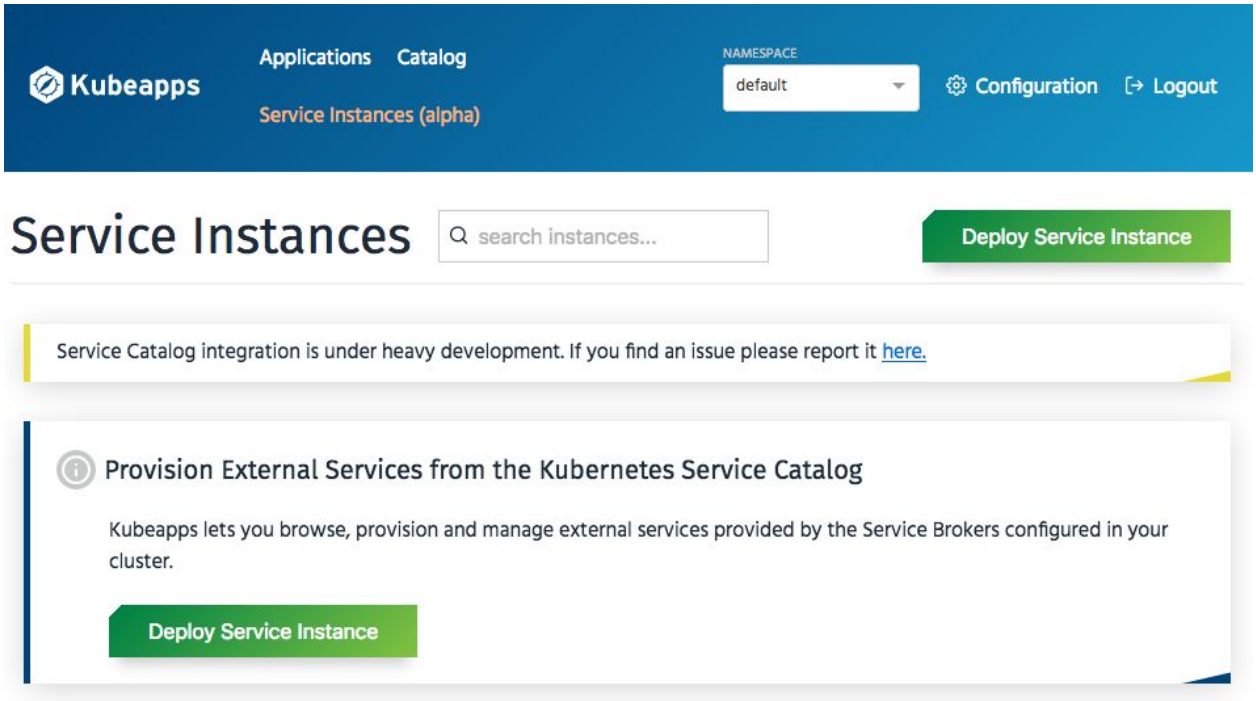
```
kubectl get ClusterServiceBroker gcp-broker
```

If the Broker has been successfully installed and the catalog has been properly downloaded you should get the following output:

```
NAME         URL
STATUS   AGE
gcp-broker
https://servicebroker.googleapis.com/v1beta1/projects/bitnamigcetest2
/brokers/default   Ready    1m
```

Kubeapps Integration

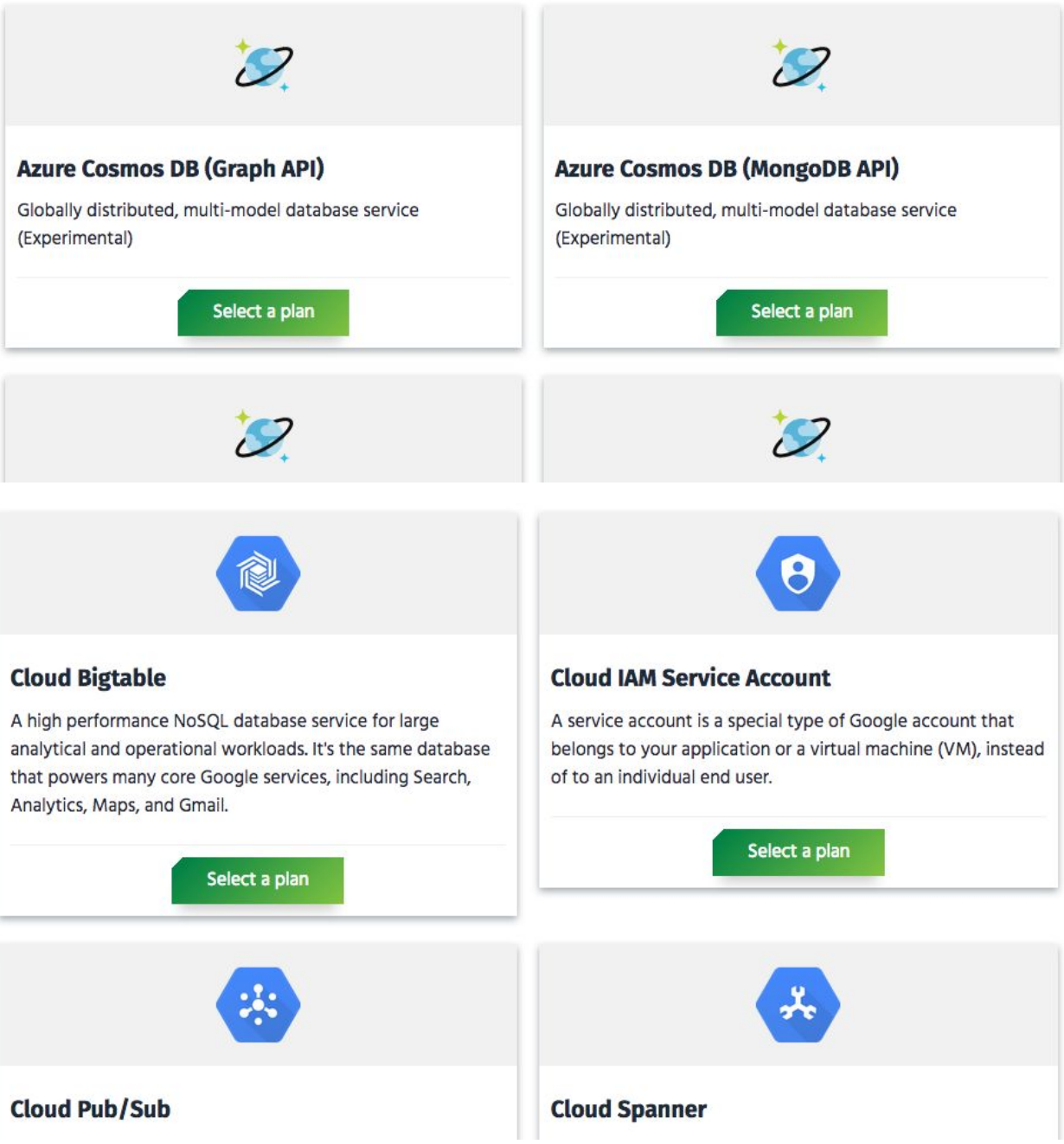
When a user clicks on Services Instances menu in Kubeapps they will get the list of Service Instances available in the selected namespace. As we haven't provisioned any yet, the list will be empty and we will get a message about provisioning an instance.



Clicking on "Deploy Service Instance" will provide us with a list of available classes:

Classes

Types of services available from all brokers

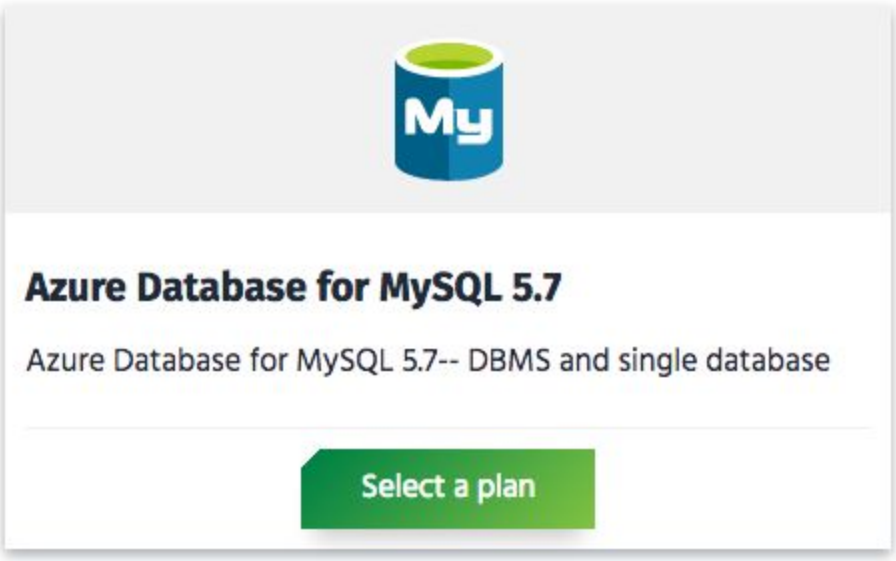


Example 1. Wordpress with Azure MySQL as Database

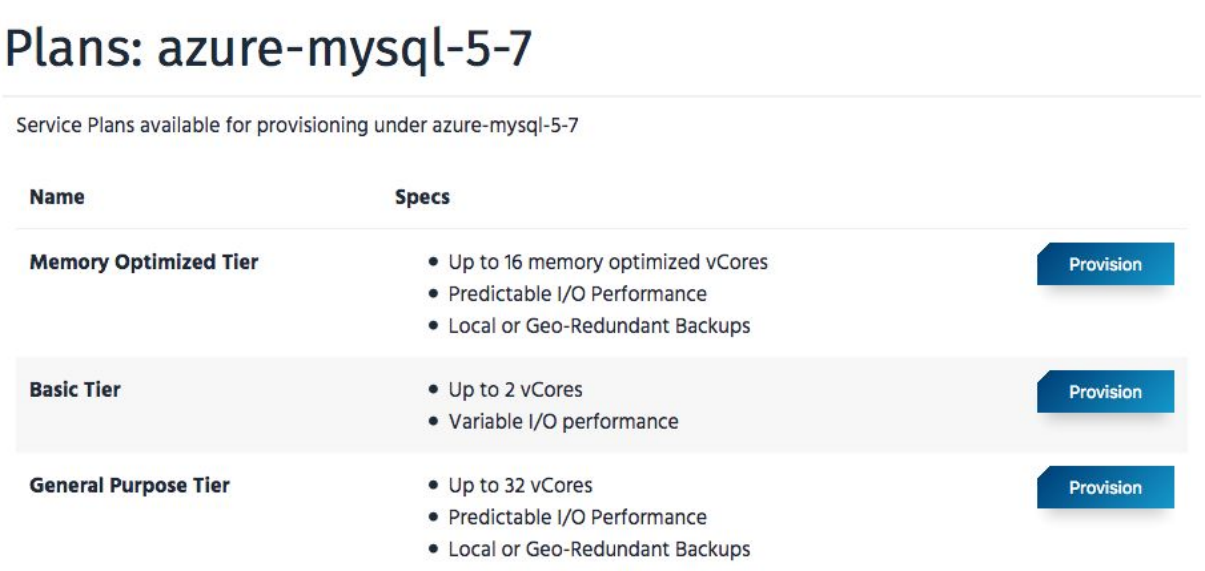
For our first example we will provision an Azure MySQL instance and we will use it as the database for our new WordPress deployment.

Provisioning the Azure MySQL Database

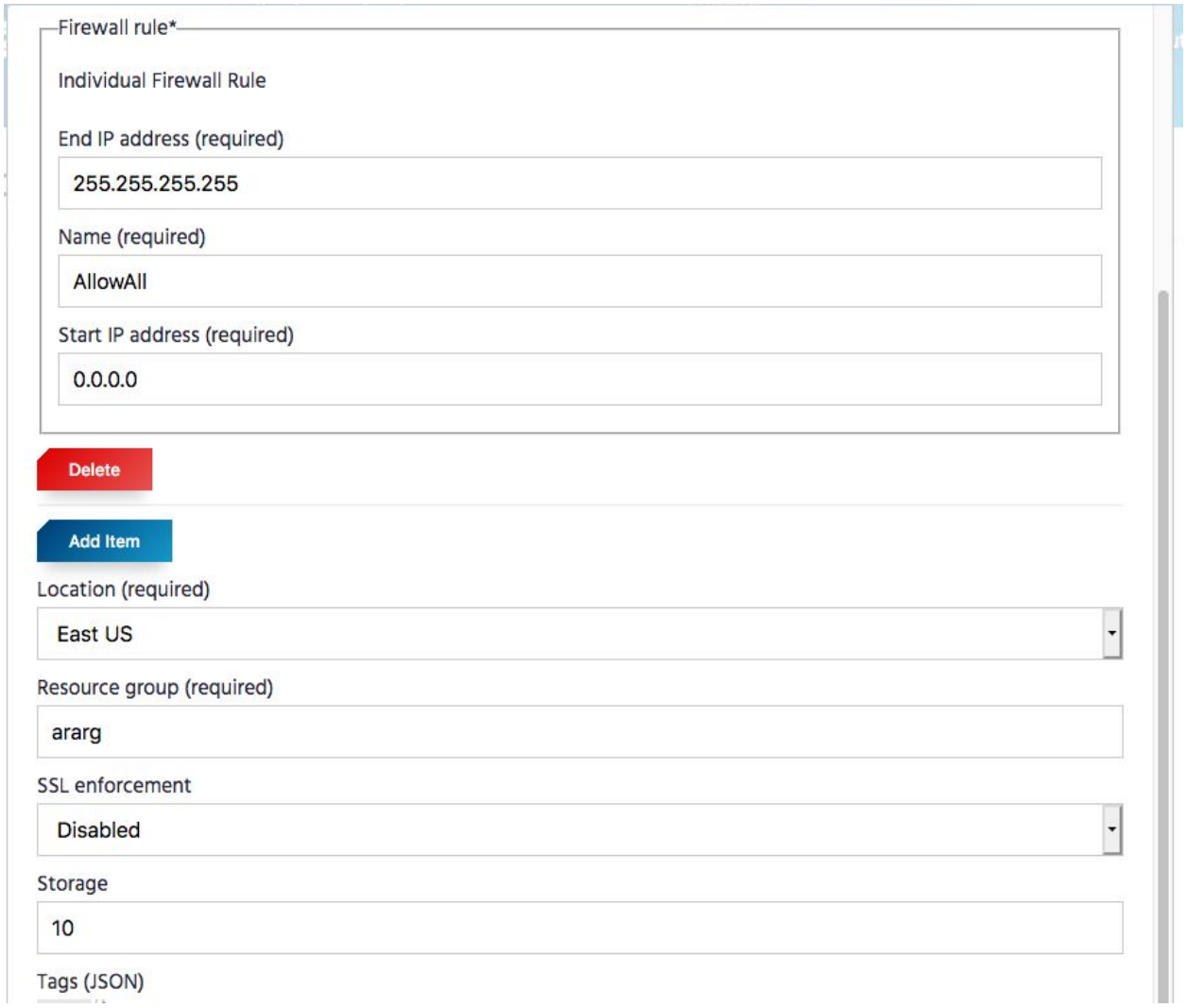
In the list of classes, click on "Select a plan" in the Azure Database for MySQL 5.7 card:



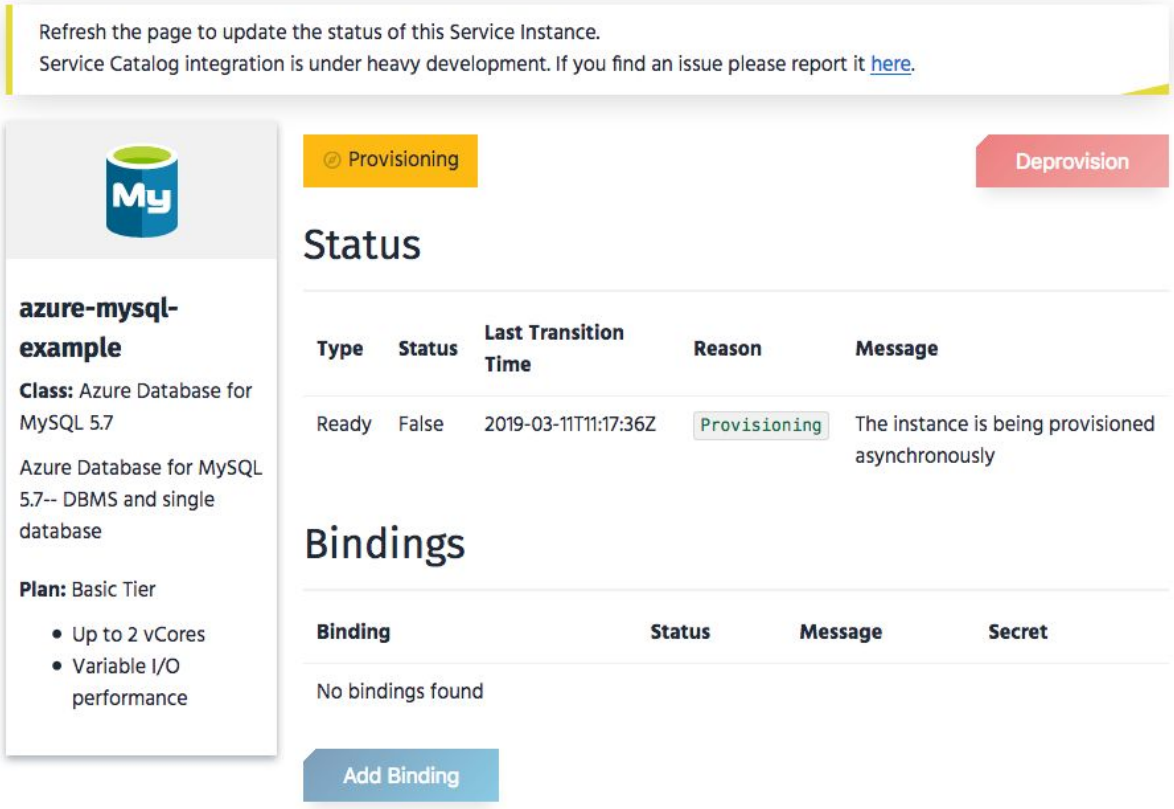
We will select the “Basic Tier” which will be enough for our testing purposes:



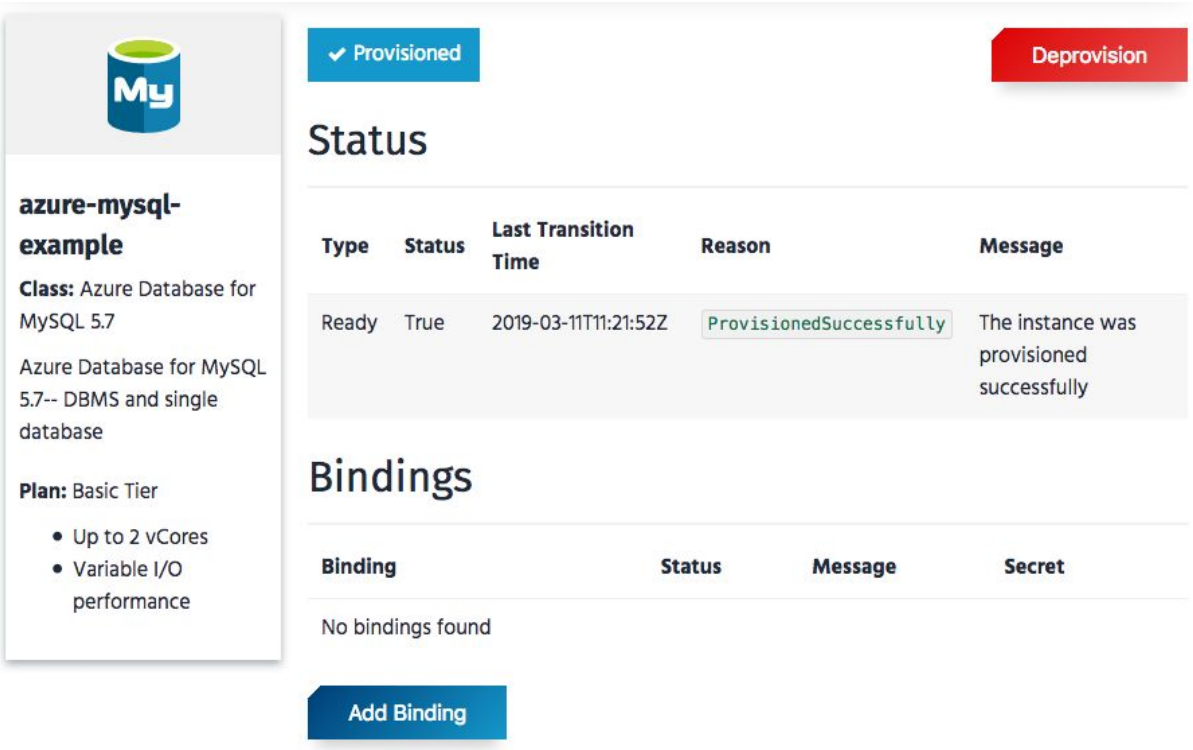
After giving it a name, we will provide a set of options for our instance, like firewall rules to allow incoming traffic, the location of the instance, and the Azure resource group:



Once we click on “Submit” the instance will start provisioning. Refresh the page regularly to check on the progress. Provisioning an Azure MySQL instance takes around 10-15 minutes.



Once provisioned you will be able to create a Binding to connect your application to it. Click on “Add Binding” and select a name for it (the default should be OK). Creating a new binding should be very fast.



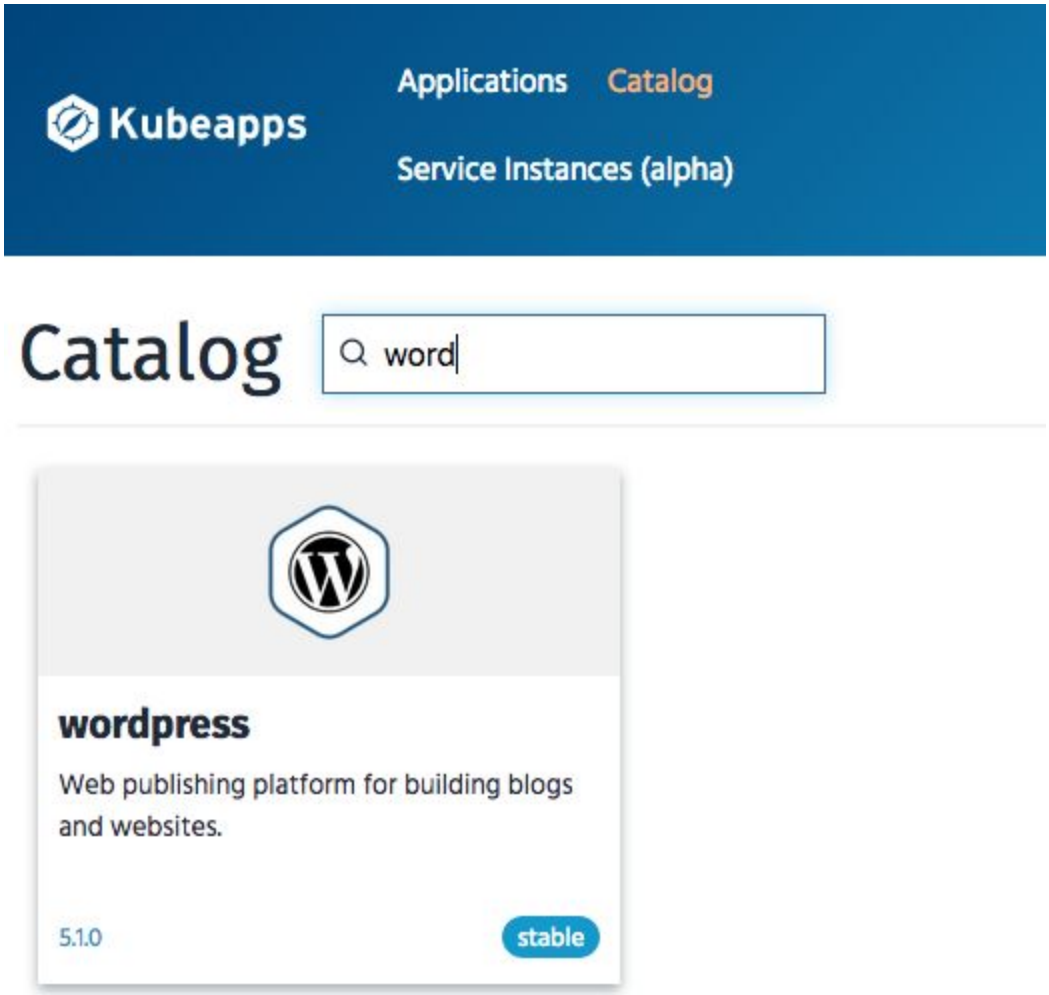
Once the binding creation has been completed, the details of the created secret can be explored directly from the same page, click on "Show". (Make sure to have access to this page (maybe in a different browser tab) as we will need those values in the next step).

It is important to understand the schema of the secret, as it is dependent on the broker and the instance. For Azure MySQL the secret will have the following schema:

```
database: name of the database
host: the URL of the instance
username: the user name to connect to the database
password: the password for user username
port: the port where MySQL is listening
[...]
```

Deploying Wordpress

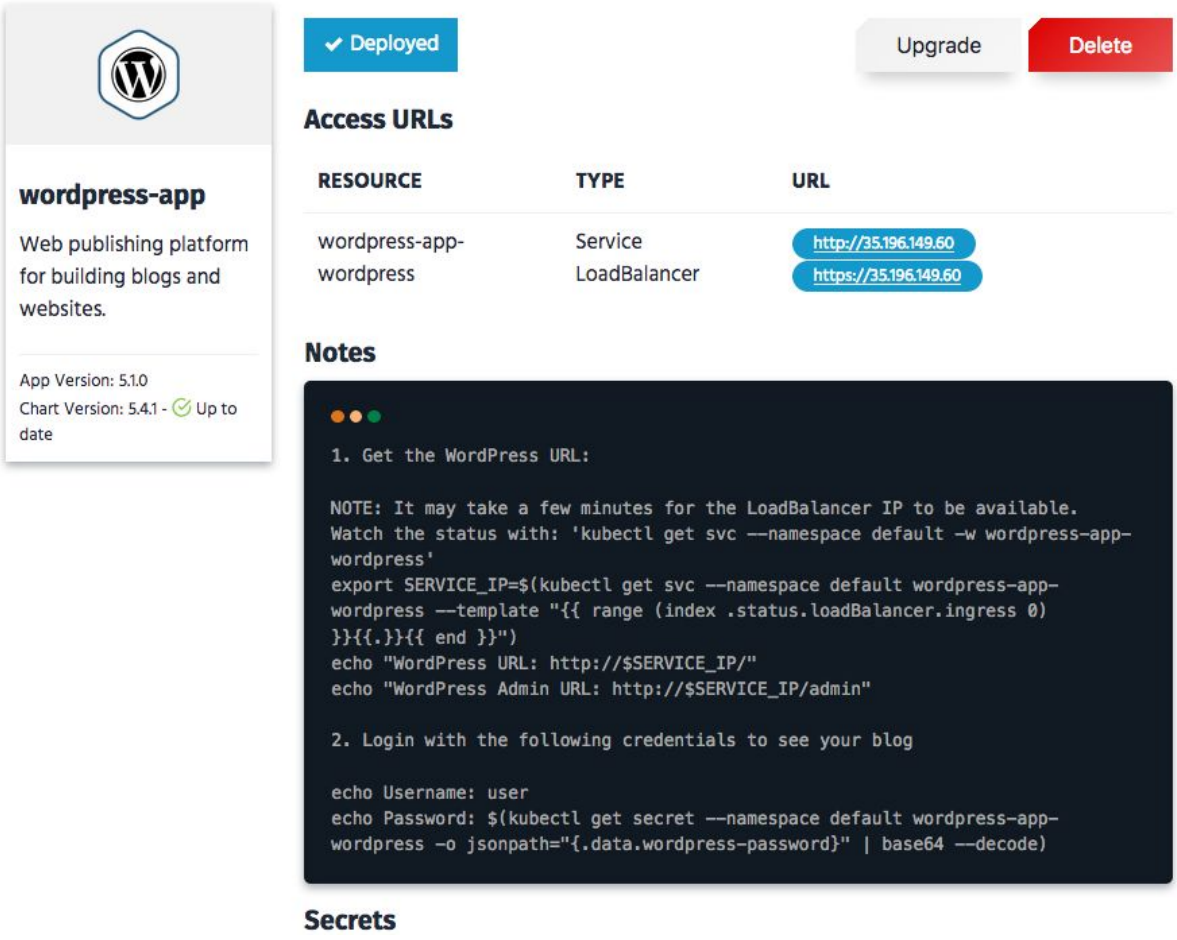
We will now deploy Wordpress, using Azure MySQL as database. In the Catalog we will search for WordPress:



We will click on Deploy and will modify the values.yaml of the application with the following values:

```
externalDatabase.host: host value in the binding secret
externalDatabase.user: username value in the binding secret
externalDatabase.password: password value in the binding secret
externalDatabase.database: database value in the binding secret
mariadb.enabled: false (to avoid deploying a MariaDB database)
```

Once those values have been modified, click on “Submit” and wait until the deployment is completed:



If we check the WordPress pod log we can see that it connected successfully to the Azure MySQL database:

```
kubectl logs wordpress-app-wordpress-597b9dbb5-2rk4k

Welcome to the Bitnami wordpress container
Subscribe to project updates by watching
https://github.com/bitnami/bitnami-docker-wordpress
Submit issues and feature requests at
https://github.com/bitnami/bitnami-docker-wordpress/issues

WARN  ==> You set the environment variable ALLOW_EMPTY_PASSWORD=yes.
For safety reasons, do not use this flag in a production environment.
nami    INFO  Initializing apache
apache  INFO  ==> Patching httpoxy...
apache  INFO  ==> Configuring dummy certificates...
nami    INFO  apache successfully initialized
nami    INFO  Initializing php
nami    INFO  php successfully initialized
nami    INFO  Initializing mysql-client
nami    INFO  mysql-client successfully initialized
nami    INFO  Initializing libphp
nami    INFO  libphp successfully initialized
nami    INFO  Initializing wordpress
mysql-c INFO  Trying to connect to MySQL server
mysql-c INFO  Found MySQL server listening at
93489418-7a54-4b2c-b807-d239cb26ad5d.mysql.database.azure.com:3306
mysql-c INFO  MySQL server listening and working at
93489418-7a54-4b2c-b807-d239cb26ad5d.mysql.database.azure.com:3306
```

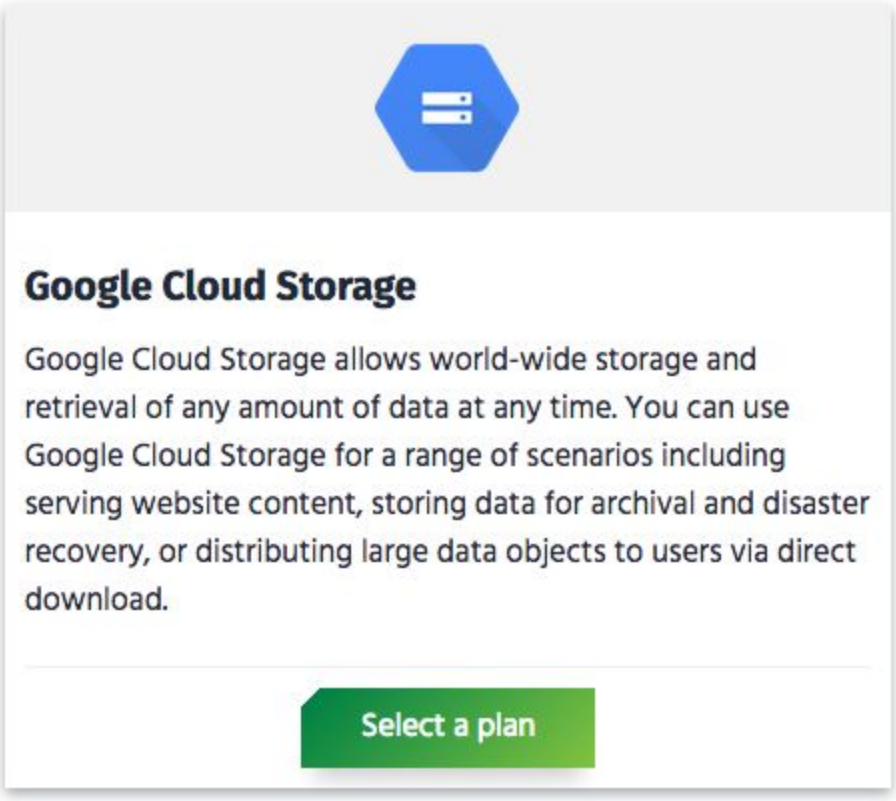
Example 2. Chartmuseum Using a GCP Storage Bucket

ChartMuseum is an open-source project that allows you to easily have your own valid Helm Chart Repository. It has support for different cloud storage backends, including Google Cloud Storage.

In this example we will create a Google Cloud Storage instance with Kubeapps Service Catalog integration, and we will use it as backend for a ChartMuseum deployment.

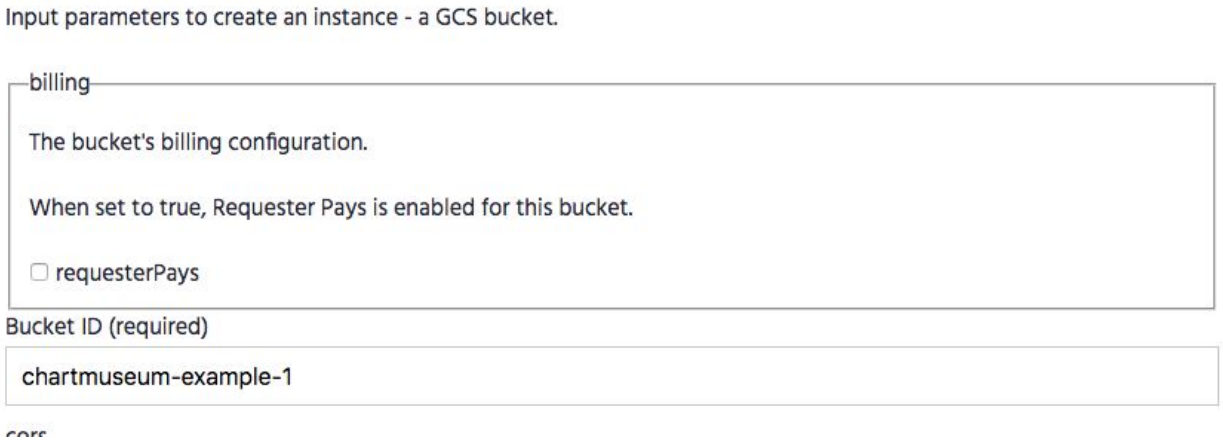
Provisioning the Google Cloud Storage bucket

Under Service Instances click on Deploy Service Instance and select Google Cloud Storage:

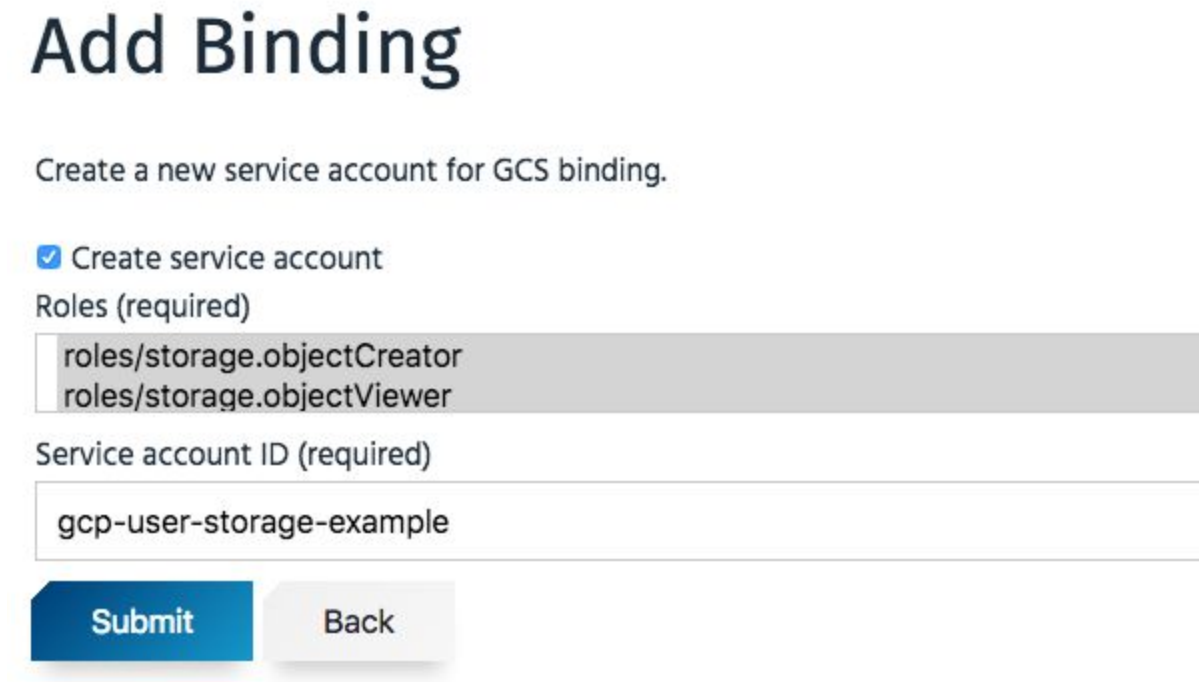


Select the Beta plan and provide a name.

We will be adding some parameters to configure the name of the bucket and the location. We will leave the rest of the options with the default values:



Once provisioned, click on Add binding to create a binding:



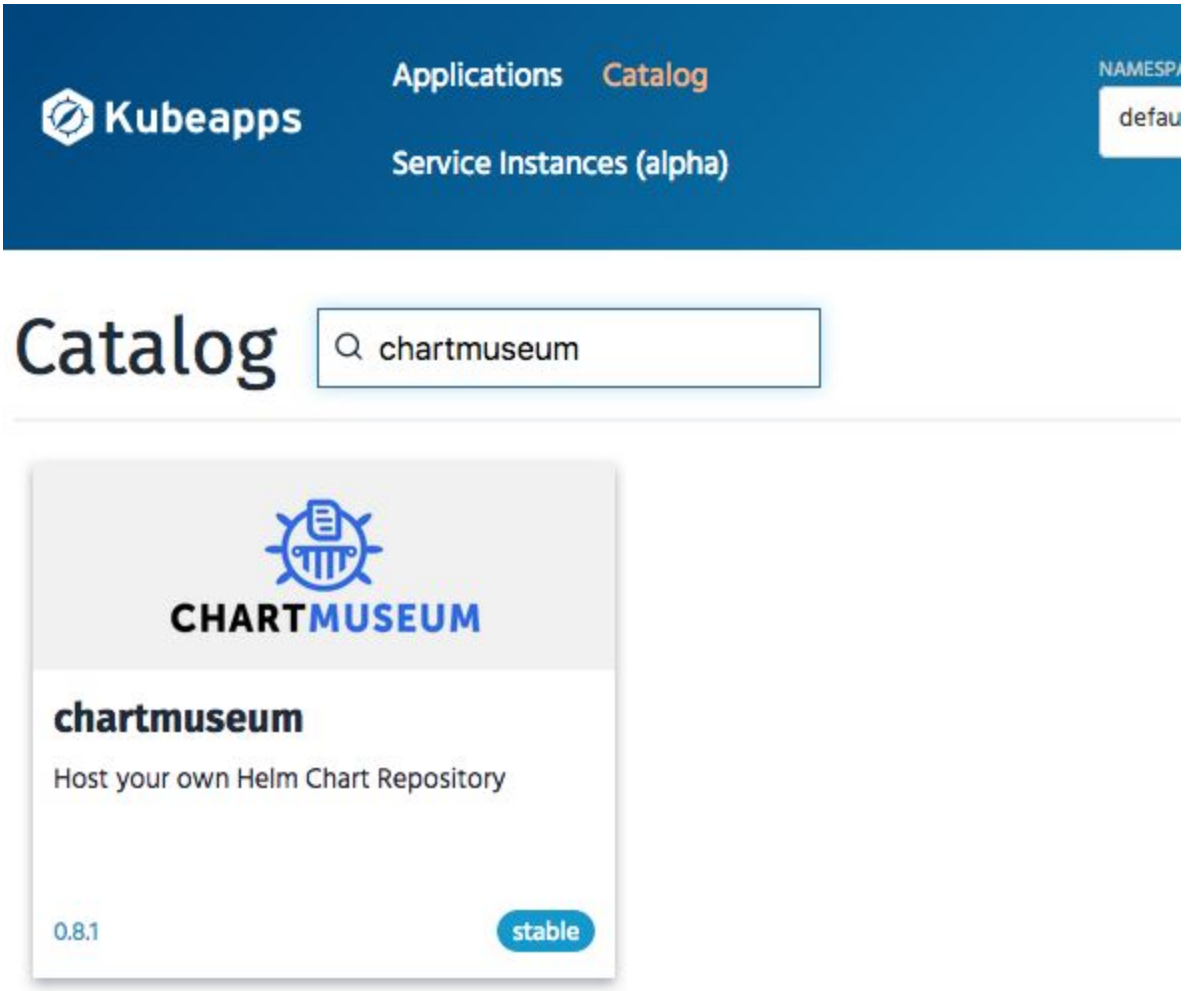
That will create a new service account in your GCP project with the right permissions to access the bucket.
Once the binding creation has been completed, the details of the created secret can be explored directly from the same page, clicking on show. (Make sure to have access to this page (maybe in a different browser tab) as we will need those values in the next step).

It is important to understand the schema of the secret, as it is dependent on the broker and the instance. For Google Storage the secret will have the following schema:

```
bucketId: the id of the bucket
privateKeyData: JSON with authentication details for the bucket
projectId: the GCP project where the bucket was created
serviceAccount: the service account to access the bucket
```

Deploy Chartmuseum

We will now deploy ChartMuseum, using Google Cloud Storage as backend. In the Catalog we will search for chartmuseum:



Click on “Deploy“ and modify the following values:

```
env.open.STORAGE: google
env.open.STORAGE_GOOGLE_BUCKET: <name of the created bucket in the
instance>
env.open.STORAGE_GOOGLE_PREFIX: charts
gcp.secret.enabled: true
gcp.secret.name: <name of the binding secret>
gcp.secret.key: privateKeyData
```

As ChartMuseum expects the format of the authentication information contained in the JSON, it is only needed to specify the key of the secret that contains that JSON (in our case `privateKeyData`).

Click on “Submit” and it should be deployed successfully, using the GCP bucket as backend.

bitnami