Zero to JupyterHub

Release 0.0.1-set.by.chartpress

Project Jupyter Contributors

May 25, 2021

1 What To Expect **3**

2 Setup a Kubernetes cluster **5**
2.1 Setup Kubernetes 5

3 Setup JupyterHub **35**
3.1 Setup JupyterHub 35

4 Customization Guide **43**
4.1 Customization Guide 43

5 Administrator Guide **57**
5.1 Administrator Guide 57

6 Resources from the community **87**
6.1 Community Resources 87

7 Reference **91**
7.1 Reference 91

8 Institutional support **117**

Index **119**

JupyterHub allows users to interact with a computing environment through a webpage. As most devices have access to a web browser, JupyterHub makes it is easy to provide and standardize the computing environment of a group of people (e.g., for a class of students or an analytics team).

This project will help you set up your own JupyterHub on a cloud and leverage the clouds scalable nature to support large groups of users. Thanks to *Kubernetes*, we are not tied to a specific cloud provider.

Note that this project is under active development so information and tools may change. *You can be a part of this change!* If you see anything that is incorrect or have any questions, feel free to reach out in the gitter chat or create an issue at the issues page. If you have tips or deployments that you would like to share, see *Resources from the community*.

This documentation is for jupyterhub chart version 0.0.1-set.by.chartpress, which deploys JupyterHub 1.1.0.

This version of the chart requires kubernetes 1.11 and helm 2.11.

CHAPTER 1

What To Expect

This guide will help you deploy and customize your own JupyterHub on a cloud. While doing this, you will gain valuable experience with:

- **A cloud provider** such Google Cloud, Microsoft Azure, Amazon EC2, IBM Cloud…
- **Kubernetes** to manage resources on the cloud
- **Helm** to configure and control the packaged JupyterHub installation
- **JupyterHub** to give users access to a Jupyter computing environment
- **A terminal interface** on some operating system

It's also possible you end up getting experienced with:

- **Docker** to build customized image for the users
- **Domain registration** to make the hub available at https://your-domain-name.com

Note: For a more elaborate introduction to the tools and services that JupyterHub depends upon, see our *Utilized Tools* page.

CHAPTER 2

Setup a Kubernetes cluster

This section describes a Kubernetes cluster and outlines how to complete *Step Zero: your Kubernetes cluster* for different cloud providers and infrastructure.

2.1 Setup Kubernetes

Kubernetes' documentation describes the many ways to set up a cluster. We attempt to provide quick instructions for the most painless and popular ways of setting up a Kubernetes cluster on various cloud providers and on other infrastructure.

Choose one option and proceed.

2.1.1 Kubernetes on Google Cloud (GKE)

Google Kubernetes Engine (GKE) is the simplest and most common way of setting up a Kubernetes Cluster. You may be able to receive free credits for trying it out (though note that a free account comes with limitations). Either way, you will need to connect your credit card or other payment method to your google cloud account.

1. Go to console.cloud.google.com and log in.

   Note: Consider setting a cloud budget for your Google Cloud account in order to make sure you don't accidentally spend more than you wish to.

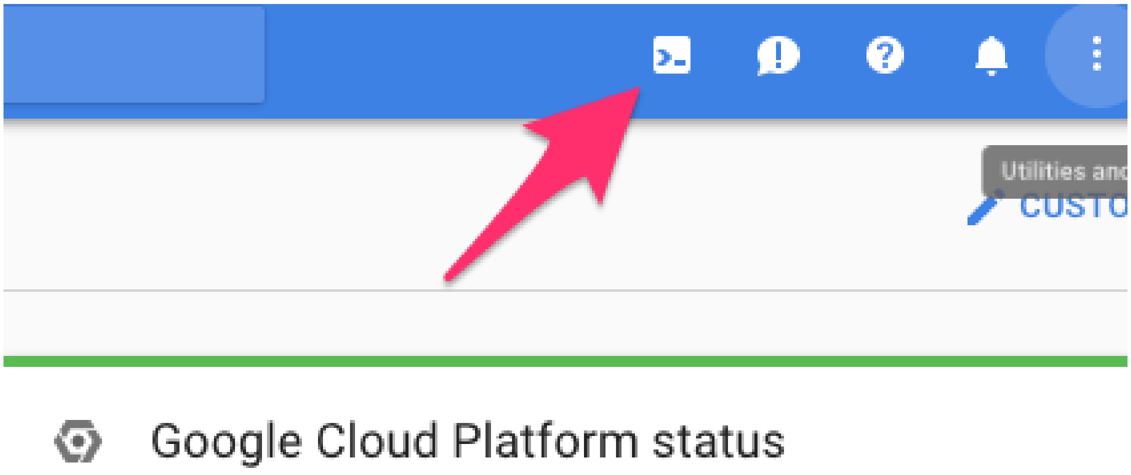
2. Go to and enable the Kubernetes Engine API.

3. Choose a terminal.

   You can either to use a web based terminal or install and run the required command line interfaces on your own computer's terminal. We recommend starting out by using the web based terminal. Choose one set of instructions below.

a. **Use a web based terminal:**

Start *Google Cloud Shell* from console.cloud.google.com by clicking the button shown below. You are now in control of a virtual machine with various tools preinstalled. If you save something in a user folder they will remain available to you if you return at a later stage. Additional documentation about Google Cloud shell is available here



b. **Use your own computer's terminal:**

1. Download and install the `gcloud` command line tool at its install page. It will help you create and communicate with a Kubernetes cluster.
2. Install `kubectl` (reads *kube control*), it is a tool for controlling Kubernetes clusters in general. From your terminal, enter:

```
gcloud components install kubectl
```

4. Create a managed Kubernetes cluster and a default node pool.

Ask Google Cloud to create a managed Kubernetes cluster and a default node pool to get nodes from. *Nodes* represents hardware and a *node pool* will keep track of how much of a certain type of hardware that you would like.

```
gcloud container clusters create \
  --machine-type n1-standard-2 \
  --num-nodes 2 \
  --zone <compute zone from the list linked below> \
  --cluster-version latest \
  <CLUSTERNAME>
```

- Replace `<CLUSTERNAME>` with a name that can be used to refer to this cluster in the future.
- `--machine-type` specifies the amount of CPU and RAM in each node within this default node pool. There is a variety of types to choose from.
- `--num-nodes` specifies how many nodes to spin up. You can change this later through the cloud console or using the `gcloud` command line tool.
- `--zone` specifies the data center zone where your cluster will be created. You can pick something from this list that is not too far away from your users.

- **Note:** A region in GCP is a geographical region with at least three zones, where each zone is representing a datacenter with servers etc.

  - A regional cluster creates pods across zones in a region(three by default), distributing Kubernetes resources across multiple zones in the region. This is different from the default cluster, which has all its resources within a single zone(as shown above).
  - A regional cluster has Highly Available (HA) kubernetes api-servers, this allows jupyterhub which uses them to have no downtime during upgrades of kubernetes itself.
  - They also increase control plane uptime to 99.95%.
  - To avoid tripling the number of nodes while still having HA kubernetes, the `--node-locations` flag can be used to specify a single zone to use.

5. To test if your cluster is initialized, run:

   ```
   kubectl get node
   ```

   The response should list two running nodes (or however many nodes you set with `--num-nodes` above).

6. Give your account permissions to perform all administrative actions needed.

   ```
   kubectl create clusterrolebinding cluster-admin-binding \
     --clusterrole=cluster-admin \
     --user=<GOOGLE-EMAIL-ACCOUNT>
   ```

   Replace `<GOOGLE-EMAIL-ACCOUNT>` with the exact email of the Google account you used to sign up for Google Cloud.

   Note: Did you enter your email correctly? If not, you can run `kubectl delete clusterrolebinding cluster-admin-binding` and do it again.

7. [optional] Create a node pool for users

   This is an optional step, for those who want to separate user pods from "core" pods such as the Hub itself and others. See *Optimizations* for details on using a dedicated user node pool.

   The nodes in this node pool are for the users only. The node pool has autoscaling enabled along with a lower and an upper scaling limit. This means that the amount of nodes is automatically adjusted along with the amount of users scheduled.

   The `n1-standard-2` machine type has 2 CPUs and 7.5 GB of RAM each of which about 0.2 CPU will be requested by system pods. It is a suitable choice for a free account that has a limit on a total of 8 CPU cores.

   Note that the node pool is *tainted*. Only user pods that are configured with a *toleration* for this taint can schedule on the node pool's nodes. This is done in order to ensure the autoscaler will be able to scale down when the user pods have stopped.

   ```
   gcloud beta container node-pools create user-pool \
     --machine-type n1-standard-2 \
     --num-nodes 0 \
     --enable-autoscaling \
     --min-nodes 0 \
     --max-nodes 3 \
     --node-labels hub.jupyter.org/node-purpose=user \
   ```

(continues on next page)

(continued from previous page)

```
    --node-taints hub.jupyter.org_dedicated=user:NoSchedule \
    --zone us-central1-b \
    --cluster <CLUSTERNAME>
```

Congrats. Now that you have your Kubernetes cluster running, it's time to begin *Setup JupyterHub*.

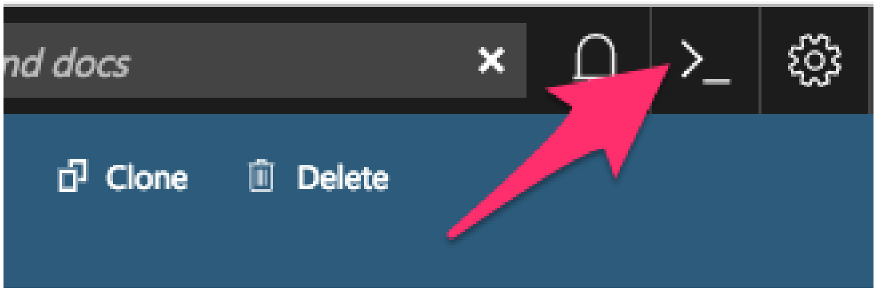
2.1.2 Kubernetes on Microsoft Azure Kubernetes Service (AKS)

You can create a Kubernetes cluster either through the Azure portal website, or using the Azure command line tools.

This page describes the commands required to setup a Kubernetes cluster using the command line. If you prefer to use the Azure portal see the Azure Kubernetes Service quickstart.

1. Prepare your Azure shell environment. You have two options, one is to use the Azure interactive shell, the other is to install the Azure command-line tools locally. Instructions for each are below.

   - **Using the Azure interactive shell**. The Azure Portal contains an interactive shell that you can use to communicate with your Kubernetes cluster. To access this shell, go to portal.azure.com and click on the button below.

     Note:

     - If you get errors like `could not retrieve token from local cache`, try refreshing your browser window.
     - The first time you do this, you'll be asked to create a storage account where your shell filesystem will live.

   - **Install command-line tools locally**. You can access the Azure CLI via a package that you can install locally.

     To do so, first follow the installation instructions in the Azure documentation. Then run the following command to connect your local CLI with your account:

     ```
     az login
     ```

     You'll need to open a browser and follow the instructions in your terminal to log in.

2. Activate the correct subscription. Azure uses the concept of **subscriptions** to manage spending. You can get a list of subscriptions your account has access to by running:

   ```
   az account list --refresh --output table
   ```

   Pick the subscription you want to use for creating the cluster, and set that as your default. If you only have one subscription you can ignore this step.

```
az account set --subscription <YOUR-CHOSEN-SUBSCRIPTION-NAME>
```

3. Create a resource group. Azure uses the concept of **resource groups** to group related resources together. We need to create a resource group in a given data center location. We will create computational resources *within* this resource group.

   ```
   az group create \
                 --name=<RESOURCE-GROUP-NAME> \
                 --location=centralus \
                 --output table
   ```

   where:

   - `--name` specifies the name of your resource group. We recommend using something that uniquely identifies this hub. For example, if you are creating a resource group for UC Berkeley's 2018 Spring Data100 Course, you may give it a `<RESOURCE-GROUP-NAME>` of `ucb_2018sp_data100_hub`.
   - `--location` specifies the location of the data center you want your resource to be in. In this case, we used the `centralus` location. For other options, see the Azure list of locations that support AKS.
   - `--output table` specifies that the output should be in human readable format, rather than the default JSON output. We shall use this with most commands when executing them by hand.

   Note: Consider setting a cloud budget for your Azure account in order to make sure you don't accidentally spend more than you wish to.

4. Choose a cluster name.

   In the following steps we'll run commands that ask you to input a cluster name. We recommend using something descriptive and short. We'll refer to this as `<CLUSTER-NAME>` for the remainder of this section.

   The next step will create a few files on your filesystem, so first create a folder in which these files will go. We recommend giving it the same name as your cluster:

   ```
   mkdir <CLUSTER-NAME>
   cd <CLUSTER-NAME>
   ```

5. Create an ssh key to secure your cluster.

   ```
   ssh-keygen -f ssh-key-<CLUSTER-NAME>
   ```

   It will prompt you to add a password, which you can leave empty if you wish. This will create a public key named `ssh-key-<CLUSTER-NAME>.pub` and a private key named `ssh-key-<CLUSTER-NAME>`. Make sure both go into the folder we created earlier, and keep both of them safe!

   Note: This command will also print out something to your terminal screen. You don't need to do anything with this text.

6. Create a virtual network and sub-network.

   Kubernetes does not by default come with a controller that enforces `networkpolicy` resources. `networkpolicy` resources are important as they define how Kubernetes pods can securely communicate with one another and the outside sources, for example, the internet.

   To enable this in Azure, we must first create a Virtual Network with Azure's own network policies enabled.

This section of the documentation is following the Microsoft Azure tutorial on creating an AKS cluster and enabling network policy, which includes information on using Calico network policies.

```
az network vnet create \
    --resource-group <RESOURCE-GROUP-NAME> \
    --name <VNET-NAME> \
    --address-prefixes 10.0.0.0/8 \
    --subnet-name <SUBNET-NAME> \
    --subnet-prefix 10.240.0.0/16
```

where:

- `--resource-group` is the ResourceGroup you created
- `--name` is the name you want to assign to your virtual network, for example, `hub-vnet`
- `--address-prefixes` are the IP address prefixes for your virtual network
- `--subnet-name` is your desired name for your subnet, for example, `hub-subnet`
- `--subnet-prefixes` are the IP address prefixes in CIDR format for the subnet

We will now retrieve the application IDs of the VNet and subnet we just created and save them to bash variables.

```
VNET_ID=$(az network vnet show \
    --resource-group <RESOURCE-GROUP-NAME> \
    --name <VNET-NAME> \
    --query id \
    --output tsv)
SUBNET_ID=$(az network vnet subnet show \
    --resource-group <RESOURCE-GROUP-NAME> \
    --vnet-name <VNET-NAME> \
    --name <SUBNET-NAME> \
    --query id \
    --output tsv)
```

We will create an Azure Active Directory (Azure AD) service principal for use with the cluster, and assign the Contributor role for use with the VNet. Make sure `SERVICE-PRINCIPAL-NAME` is something recognisable, for example, `binderhub-sp`.

```
SP_PASSWD=$(az ad create-for-rbac \
    --name <SERVICE-PRINCIPAL-NAME> \
    --role Contributor \
    --scopes $VNET_ID \
    --query password \
    --output tsv)
SP_ID=$(az ad sp show \
    --id http://<SERVICE-PRINCIPAL-NAME> \
    --query appId \
    --output tsv)
```

Warning: You will need Owner role on your subscription for this step to succeed.

7. Create an AKS cluster.

   The following command will request a Kubernetes cluster within the resource group that we created earlier.

```
az aks create \
    --name <CLUSTER-NAME> \
    --resource-group <RESOURCE-GROUP-NAME> \
    --ssh-key-value ssh-key-<CLUSTER-NAME>.pub \
    --node-count 3 \
    --node-vm-size Standard_D2s_v3 \
    --service-principal $SP_ID \
    --client-secret $SP_PASSWD \
    --dns-service-ip 10.0.0.10 \
    --docker-bridge-address 172.17.0.1/16 \
    --network-plugin azure \
    --network-policy azure \
    --service-cidr 10.0.0.0/16 \
    --vnet-subnet-id $SUBNET_ID \
    --output table
```

where:

- `--name` is the name you want to use to refer to your cluster
- `--resource-group` is the ResourceGroup you created
- `--ssh-key-value` is the ssh public key created
- `--node-count` is the number of nodes you want in your Kubernetes cluster
- `--node-vm-size` is the size of the nodes you want to use, which varies based on what you are using your cluster for and how much RAM/CPU each of your users need. There is a list of all possible node sizes for you to choose from, but not all might be available in your location. If you get an error whilst creating the cluster you can try changing either the region or the node size.
- `--service-principal` is the application ID of the service principal we created
- `--client-secret` is the password for the service principal we created
- `--dns-service-ip` is an IP address assigned to the Kubernetes DNS service
- `--docker-bridge-address` is a specific IP address and netmask for the Docker bridge, using standard CIDR notation
- `--network-plugin` is the Kubernetes network plugin to use. In this example, we have used Azure's own implementation.
- `--network-policy` is the Kubernetes network policy to use. In this example, we have used Azure's own implementation.
- `--service-cidr` is a CIDR notation IP range from which to assign service cluster IPs
- `vnet-subnet-id` is the application ID of the subnet we created
- This command will install the default version of Kubernetes. You can pass `--kubernetes-version` to install a different version.

This should take a few minutes and provide you with a working Kubernetes cluster!

8. If you're using the Azure CLI locally, install kubectl, a tool for accessing the Kubernetes API from the commandline:

```
az aks install-cli
```

Note: kubectl is already installed in Azure Cloud Shell.

9. Get credentials from Azure for `kubectl` to work:

```
az aks get-credentials \
             --name <CLUSTER-NAME> \
             --resource-group <RESOURCE-GROUP-NAME> \
             --output table
```

where:

- `--name` is the name you gave your cluster
- `--resource-group` is the ResourceGroup you created

This automatically updates your Kubernetes client configuration file.

10. Check if your cluster is fully functional

```
kubectl get node
```

The response should list three running nodes and their Kubernetes versions! Each node should have the status of `Ready`, note that this may take a few moments.

Note: If you create the cluster using the Azure Portal you must enable RBAC. RBAC is enabled by default when using the command line tools.

Congrats. Now that you have your Kubernetes cluster running, it's time to begin *Setup JupyterHub*.

2.1.3 Kubernetes on Microsoft Azure Kubernetes Service (AKS) with Autoscaling

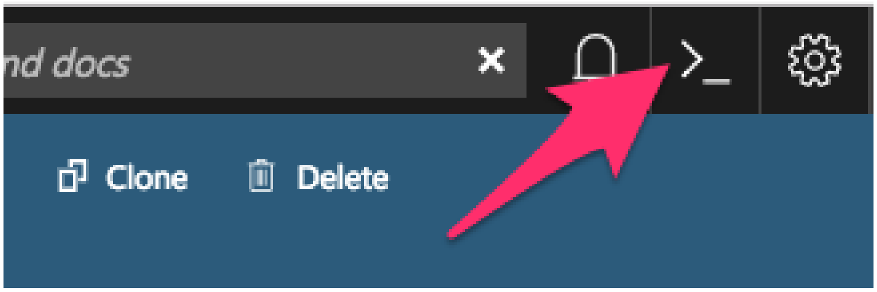
Warning: These instructions involve parts of the Azure command line that are in preview, hence the following documentation is subject to change.

You can create a Kubernetes cluster either through the Azure portal website, or using the Azure command line tools.

This page describes the commands required to setup a Kubernetes cluster using the command line. If you prefer to use the Azure portal see the Azure Kubernetes Service quickstart.

1. Prepare your Azure shell environment. You have two options, one is to use the Azure interactive shell, the other is to install the Azure command-line tools locally. Instructions for each are below.

   - **Using the Azure interactive shell**. The Azure Portal contains an interactive shell that you can use to communicate with your Kubernetes cluster. To access this shell, go to portal.azure.com and click on the button below.



Note:

- If you get errors like `could not retrieve token from local cache`, try refreshing your browser window.
- The first time you do this, you'll be asked to create a storage account where your shell filesystem will live.

- **Install command-line tools locally**. You can access the Azure CLI via a package that you can install locally.

  To do so, first follow the installation instructions in the Azure documentation. Then run the following command to connect your local CLI with your account:

  ```
  az login
  ```

  You'll need to open a browser and follow the instructions in your terminal to log in.

2. Activate the correct subscription. Azure uses the concept of **subscriptions** to manage spending. You can get a list of subscriptions your account has access to by running:

   ```
   az account list --refresh --output table
   ```

   Pick the subscription you want to use for creating the cluster, and set that as your default. If you only have one subscription you can ignore this step.

   ```
   az account set --subscription <YOUR-CHOSEN-SUBSCRIPTION-NAME>
   ```

3. Setup the CLI for Autoscaling features. First install the aks-preview CLI extension. This will grant access to new commands.

   ```
   az extension add --name aks-preview
   ```

   We then need to register the scale set feature.

   ```
   az feature register --name VMSSPreview --namespace Microsoft.ContainerService
   ```

   A VMSS is a Virtual Machine Scale Set, that is to say an autoscalable set of virtual machines.

   The previous command will take a while to register. Use the following command to check it's status.

   ```
   az feature list \
     --output table \
     --query  "[?contains(name, 'Microsoft.ContainerService/VMSSPreview')].
   ↪{Name:name,State:properties.state}"
   ```

   Once the VMSSPreview feature has been registered, refresh the registration with the following command.

   ```
   az provider register --namespace Microsoft.ContainerService
   ```

4. Create a resource group. Azure uses the concept of **resource groups** to group related resources together. We need to create a resource group in a given data center location. We will create computational resources *within* this resource group.

   ```
   az group create \
                 --name=<RESOURCE-GROUP-NAME> \
                 --location=<LOCATION> \
                 --output table
   ```

   where:

- `--name` specifies the name of your resource group. We recommend using something that uniquely identifies this hub. For example, if you are creating a resource group for UC Berkeley's 2018 Spring Data100 Course, you may give it a `<RESOURCE-GROUP-NAME>` of `ucb_2018sp_data100_hub`.
- `--location` specifies the location of the data center you want your resource to be in. For options, see the Azure list of locations that support AKS.
- `--output table` specifies that the output should be in human readable format, rather than the default JSON output. We shall use this with most commands when executing them by hand.

Note: Consider setting a cloud budget for your Azure account in order to make sure you don't accidentally spend more than you wish to.

5. Choose a cluster name.

   In the following steps we'll run commands that ask you to input a cluster name. We recommend using something descriptive and short. We'll refer to this as `<CLUSTER-NAME>` for the remainder of this section.

   The next step will create a few files on your filesystem, so first create a folder in which these files will go. We recommend giving it the same name as your cluster:

```
mkdir <CLUSTER-NAME>
cd <CLUSTER-NAME>
```

6. Create an ssh key to secure your cluster.

```
ssh-keygen -f ssh-key-<CLUSTER-NAME>
```

   It will prompt you to add a password, which you can leave empty if you wish. This will create a public key named `ssh-key-<CLUSTER-NAME>.pub` and a private key named `ssh-key-<CLUSTER-NAME>`. Make sure both go into the folder we created earlier, and keep both of them safe!

Note: This command will also print out something to your terminal screen. You don't need to do anything with this text.

7. Create a virtual network and sub-network.

   Kubernetes does not by default come with a controller that enforces `networkpolicy` resources. `networkpolicy` resources are important as they define how Kubernetes pods can securely communicate with one another and the outside sources, for example, the internet.

   To enable this in Azure, we must first create a Virtual Network with Azure's own network policies enabled.

   This section of the documentation is following the Microsoft Azure tutorial on creating an AKS cluster and enabling network policy, which includes information on using Calico network policies.

```
az network vnet create \
    --resource-group <RESOURCE-GROUP-NAME> \
    --name <VNET-NAME> \
    --address-prefixes 10.0.0.0/8 \
    --subnet-name <SUBNET-NAME> \
    --subnet-prefix 10.240.0.0/16
```

   where:

   - `--resource-group` is the ResourceGroup you created
   - `--name` is the name you want to assign to your virtual network, for example, `hub-vnet`

- `--address-prefixes` are the IP address prefixes for your virtual network
- `--subnet-name` is your desired name for your subnet, for example, `hub-subnet`
- `--subnet-prefixes` are the IP address prefixes in CIDR format for the subnet

We will now retrieve the application IDs of the VNet and subnet we just created and save them to bash variables.

```
VNET_ID=$(az network vnet show \
    --resource-group <RESOURCE-GROUP-NAME> \
    --name <VNET-NAME> \
    --query id \
    --output tsv)
SUBNET_ID=$(az network vnet subnet show \
    --resource-group <RESOURCE-GROUP-NAME> \
    --vnet-name <VNET-NAME> \
    --name <SUBNET-NAME> \
    --query id \
    --output tsv)
```

We will create an Azure Active Directory (Azure AD) service principal for use with the cluster, and assign the Contributor role for use with the VNet. Make sure `SERVICE-PRINCIPAL-NAME` is something recognisable, for example, `binderhub-sp`.

```
SP_PASSWD=$(az ad sp create-for-rbac \
    --name <SERVICE-PRINCIPAL-NAME> \
    --role Contributor \
    --scope $VNET_ID \
    --query password \
    --output tsv)
SP_ID=$(az ad sp show \
    --id http://<SERVICE-PRINCIPAL-NAME> \
    --query appId \
    --output tsv)
```

Warning: You will need Owner role on your subscription for this step to succeed.

8. Create an AKS cluster.

   The following command will request a Kubernetes cluster within the resource group that we created earlier.

```
az aks create --name <CLUSTER-NAME> \
              --resource-group <RESOURCE-GROUP-NAME> \
              --ssh-key-value ssh-key-<CLUSTER-NAME>.pub \
              --node-count 3 \
              --node-vm-size Standard_D2s_v3 \
              --enable-vmss \
              --enable-cluster-autoscaler \
              --min-count 3 \
              --max-count 6 \
              --kubernetes-version 1.12.7 \
              --service-principal $SP_ID \
              --client-secret $SP_PASSWD \
              --dns-service-ip 10.0.0.10 \
              --docker-bridge-address 172.17.0.1/16 \
              --network-plugin azure \
              --network-policy azure \
```

(continues on next page)

(continued from previous page)

```
              --service-cidr 10.0.0.0/16 \
              --vnet-subnet-id $SUBNET_ID \
              --output table
```

where:

- `--name` is the name you want to use to refer to your cluster
- `--resource-group` is the ResourceGroup you created
- `--ssh-key-value` is the ssh public key created
- `--node-count` is the number of nodes you want in your Kubernetes cluster
- `--node-vm-size` is the size of the nodes you want to use, which varies based on what you are using your cluster for and how much RAM/CPU each of your users need. There is a list of all possible node sizes for you to choose from, but not all might be available in your location. If you get an error whilst creating the cluster you can try changing either the region or the node size.
- `--enable-vmss` deploys the cluster as a scale set.
- `--enable-cluster-autoscaler` installs a Cluster Autoscaler onto the cluster (though counterintuitively, does not enable it!).
- `--min-count`/`--max-count` are the minimum/maximum number of nodes in the cluster at any time.
- `--kubernetes-version` installs a specific version of Kubernetes onto the cluster. To autoscale, we require `>= v 1.12.4`, though it's recommended to use the most recent version available.
- `--service-principal` is the application ID of the service principal we created
- `--client-secret` is the password for the service principal we created
- `--dns-service-ip` is an IP address assigned to the Kubernetes DNS service
- `--docker-bridge-address` is a specific IP address and netmask for the Docker bridge, using standard CIDR notation
- `--network-plugin` is the Kubernetes network plugin to use. In this example, we have used Azure's own implementation.
- `--network-policy` is the Kubernetes network policy to use. In this example, we have used Azure's own implementation.
- `--service-cidr` is a CIDR notation IP range from which to assign service cluster IPs
- `vnet-subnet-id` is the application ID of the subnet we created

This should take a few minutes and provide you with a working Kubernetes cluster!

9. If you're using the Azure CLI locally, install kubectl, a tool for accessing the Kubernetes API from the commandline:

```
az aks install-cli
```

Note: kubectl is already installed in Azure Cloud Shell.

10. Get credentials from Azure for `kubectl` to work:

```
az aks get-credentials \
             --name <CLUSTER-NAME> \
             --resource-group <RESOURCE-GROUP-NAME> \
             --output table
```

where:

- `--name` is the name you gave your cluster
- `--resource-group` is the ResourceGroup you created

This automatically updates your Kubernetes client configuration file.

11. Check if your cluster is fully functional

```
kubectl get node
```

The response should list three running nodes and their Kubernetes versions! Each node should have the status of `Ready`, note that this may take a few moments.

12. Enabling Autoscaling

We now move to the Azure Portal to enable autoscaling and set rules to manage the Cluster Autoscaler.

First we need to register Microsoft Insights for use on the active subscription.

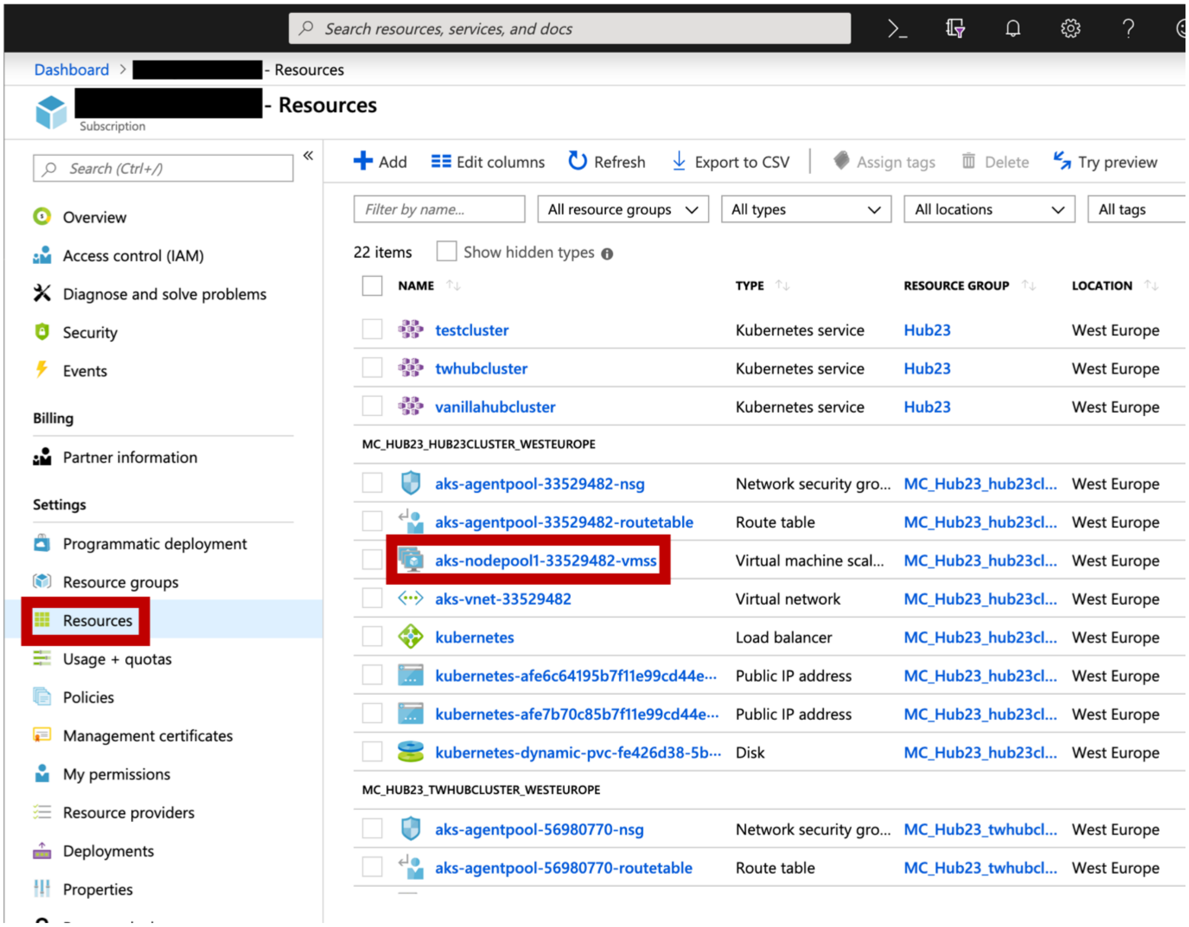
```
az provider register --namespace microsoft.insights
```

To check the status of the registration, run the following command:

```
az provider show -n microsoft.insights
```

Once the application has been registered, navigate to your active subscription on the Portal.

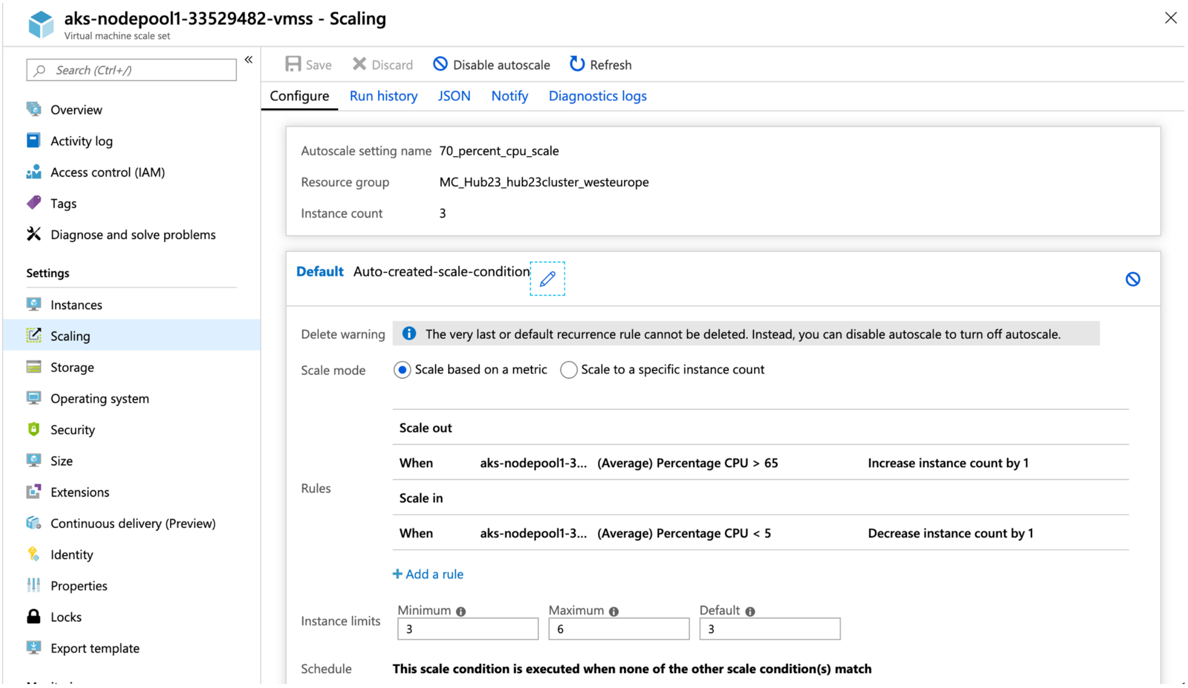
Under "Resources", select the VMSS. It should be named something like `aks-nodepool1-<random-str>-vmss`.



NAME	TYPE	RESOURCE GROUP	LOCATION
testcluster	Kubernetes service	Hub23	West Europe
twhubcluster	Kubernetes service	Hub23	West Europe
vanillahubcluster	Kubernetes service	Hub23	West Europe
MC_HUB23_HUB23CLUSTER_WESTEUROPE			
aks-agentpool-33529482-nsg	Network security gro...	MC_Hub23_hub23cl...	West Europe
aks-agentpool-33529482-routetable	Route table	MC_Hub23_hub23cl...	West Europe
aks-nodepool1-33529482-vmss	Virtual machine scal...	MC_Hub23_hub23cl...	West Europe
aks-vnet-33529482	Virtual network	MC_Hub23_hub23cl...	West Europe
kubernetes	Load balancer	MC_Hub23_hub23cl...	West Europe
kubernetes-afe6c64195b7f11e99cd44e...	Public IP address	MC_Hub23_hub23cl...	West Europe
kubernetes-afe7b70c85b7f11e99cd44e...	Public IP address	MC_Hub23_hub23cl...	West Europe
kubernetes-dynamic-pvc-fe426d38-5b...	Disk	MC_Hub23_hub23cl...	West Europe
MC_HUB23_TWHUBCLUSTER_WESTEUROPE			
aks-agentpool-56980770-nsg	Network security gro...	MC_Hub23_twhubcl...	West Europe
aks-agentpool-56980770-routetable	Route table	MC_Hub23_twhubcl...	West Europe

From the left-hand menu, select "Scaling". Click the blue "Enable autoscaling" button and an autogenerated form for a scale condition will appear. We will add two new rules to this condition:

- Increase the instance count by 1 when the average CPU usage over 10 minutes is greater than 70%
- Decrease the instance count by 1 when the average CPU usage over 10 minutes is less than 5%



Autoscale setting name 70_percent_cpu_scale

Resource group MC_Hub23_hub23cluster_westeurope

Instance count 3

Default Auto-created-scale-condition

Delete warning: The very last or default recurrence rule cannot be deleted. Instead, you can disable autoscale to turn off autoscale.

Scale mode: ● Scale based on a metric ○ Scale to a specific instance count

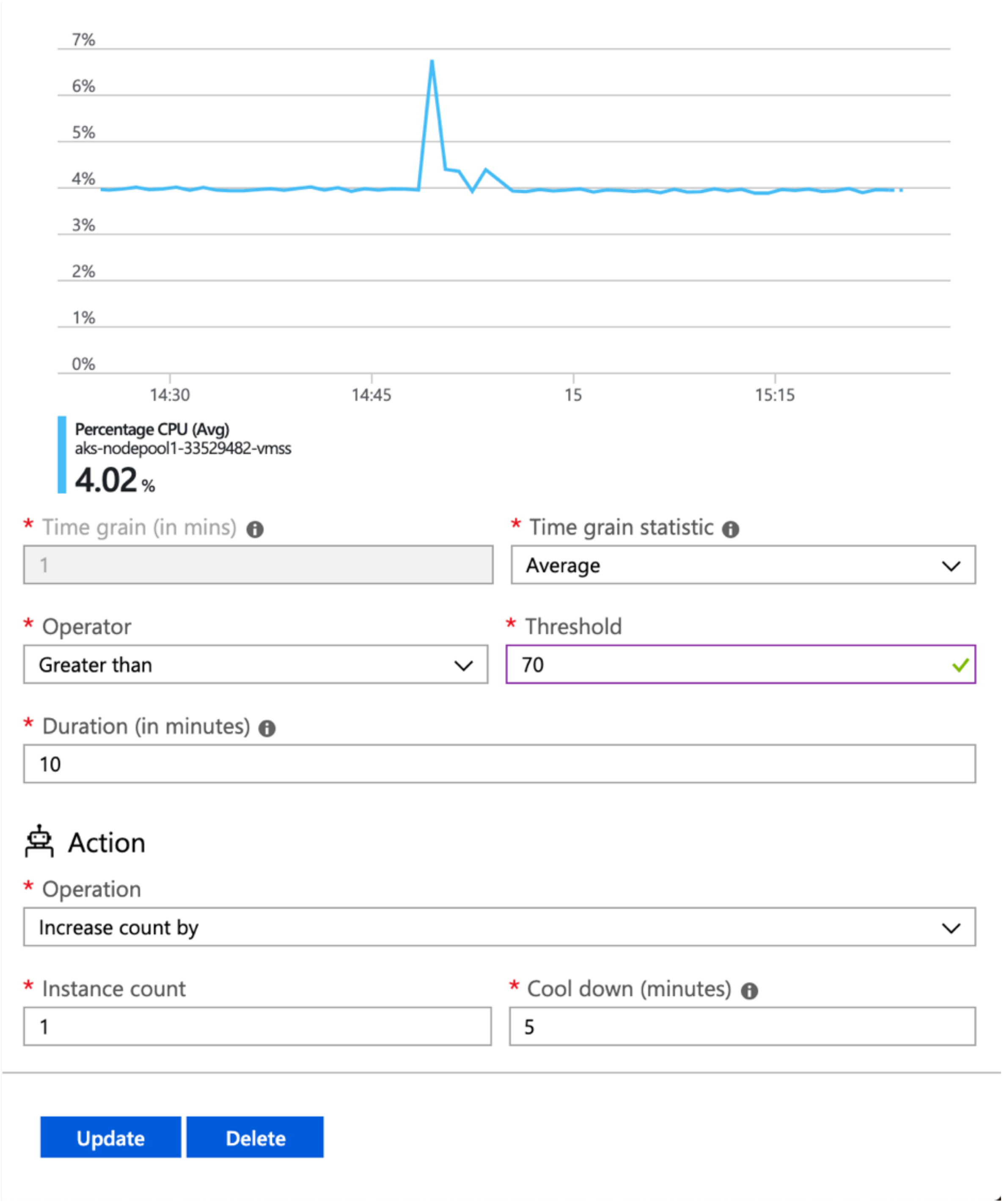
Rules

Scale out			
When	aks-nodepool1-3...	(Average) Percentage CPU > 65	Increase instance count by 1
Scale in			
When	aks-nodepool1-3...	(Average) Percentage CPU < 5	Decrease instance count by 1

Instance limits: Minimum 3, Maximum 6, Default 3

Schedule: This scale condition is executed when none of the other scale condition(s) match

Make sure the "Scale based on metric" option is selected and click "+ Add new rule", another autogenerated form will appear. This will be pre-filled with the required settings to fulfill our first rule, so save it by clicking "Update" and click "+ Add new rule" again.



Percentage CPU (Avg)
aks-nodepool1-33529482-vmss
4.02 %

* Time grain (in mins): 1

* Time grain statistic: Average

* Operator: Greater than

* Threshold: 70 ✓

* Duration (in minutes): 10

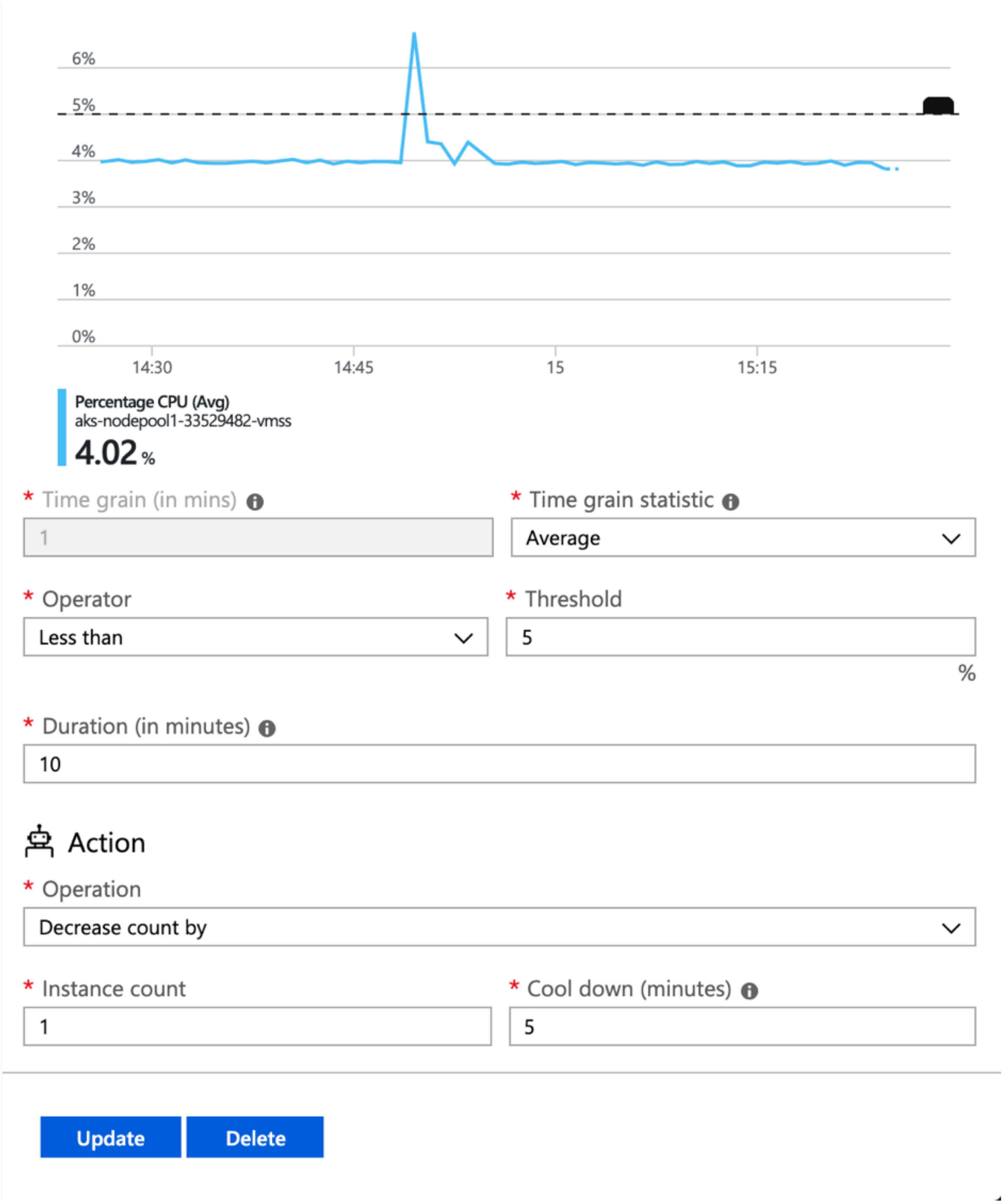
Action

* Operation: Increase count by

* Instance count: 1

* Cool down (minutes): 5

The second form needs to be edited for the second rule to decrease the instance count by 1 when the average CPU usage over 10 minutes is less than 5%. Save this rule and then save the overall scale condition, the cluster will be updated automatically.



Percentage CPU (Avg)
aks-nodepool1-33529482-vmss
4.02 %

* Time grain (in mins): 1

* Time grain statistic: Average

* Operator: Less than

* Threshold: 5 %

* Duration (in minutes): 10

Action

* Operation: Decrease count by

* Instance count: 1

* Cool down (minutes): 5

Note: This form can also be used to change `--node-count`/`--min-count`/`--max-count` that was set previously by using the "Instance limits" section of the scale condition ("Default", "Minimum" and "Maximum" respectively).

If you prefer to use the command line, you can run the following:

```
az aks update \
  --name <CLUSTER-NAME> \
```

(continues on next page)

(continued from previous page)

```
--resource-group <RESOURCE-GROUP> \
--update-cluster-autoscaler \
--min-count <DESIRED-MINIMUM-COUNT> \
--max-count <DESIRED-MAXIMUM-COUNT> \
--output table
```

Both `--min-count` and `--max-count` must be defined.

Note: If you create the cluster using the Azure Portal you must enable RBAC. RBAC is enabled by default when using the command line tools.

Congrats. Now that you have your Kubernetes cluster running, it's time to begin *Setup JupyterHub*.

2.1.4 Kubernetes on Amazon Web Services (AWS)

AWS does not have native support for Kubernetes, however there are many organizations that have put together their own solutions and guides for setting up Kubernetes on AWS.

This guide uses kops to setup a cluster on AWS. This should be seen as a rough template you will use to setup and shape your cluster.

The Procedure

1. Create a IAM Role

   This role will be used to give your CI host permission to create and destroy resources on AWS

   - AmazonEC2FullAccess
   - IAMFullAccess
   - AmazonS3FullAccess
   - AmazonVPCFullAccess
   - Route53FullAccess (Optional)

2. Create a new instance to use as your CI host. This node will deal with provisioning and tearing down the cluster.

   This instance can be small (t2.micro for example).

   When creating it, assign the IAM role created in step 1.

   Once created, download ssh keys.

3. SSH to your CI host

4. Install kops and kubectl on your CI host

   - Follow the instructions here: https://github.com/kubernetes/kops/blob/master/docs/install.md

5. Choose a cluster name

   Since we are not using pre-configured DNS we will use the suffix ".k8s.local". Per the docs, if the DNS name ends in .k8s.local the cluster will use internal hosted DNS.

   ```
   export NAME=<somename>.k8s.local
   ```

6. Setup an ssh keypair to use with the cluster

   ```
   ssh-keygen
   ```

7. Create a S3 bucket to store your cluster configuration

   Since we are on AWS we can use a S3 backing store. It is recommended to enabling versioning on the S3 bucket. We don't need to pass this into the KOPS commands. It is automatically detected by the kops tool as an env variable.

   ```
   export KOPS_STATE_STORE=s3://<your_s3_bucket_name_here>
   ```

8. Set the region to deploy in

   ```
   export REGION=`curl -s http://169.254.169.254/latest/dynamic/instance-
   ↪identity/document|grep region|awk -F\" '{print $4}'`
   ```

9. Install the AWS CLI:

   ```
   sudo apt-get update
   sudo apt-get install awscli
   ```

10. Set the availability zones for the nodes

    For this guide we will be allowing nodes to be deployed in all AZs:

```
.. code-block:: bash
export ZONES=$(aws ec2 describe-availability-zones --region $REGION | grep␣
↪ZoneName | awk '{print $2}' | tr -d '"')
```

11. Create the cluster

    For a basic setup run the following (All sizes measured in GB):

```
kops create cluster $NAME \
  --zones "$ZONES" \
  --authorization RBAC \
  --master-size t2.micro \
  --master-volume-size 10 \
  --node-size t2.medium \
  --node-volume-size 10 \
  --yes
```

    For a more secure setup add the following params to the kops command:

```
--topology private \
--networking weave \
```

    This creates a cluster where all of the masters and nodes are in private subnets and don't have external IP addresses. A mis-configured security group or insecure ssh configuration is less likely to compromise the cluster. In order to SSH into your cluster you will need to set up a bastion node. Make sure you do that step below. If you have the default number of elastic IPs (10) you may need to put in a request to AWS support to bump up that limit. The alternative is reducing the number of zones specified.

    More reading on this subject: https://github.com/kubernetes/kops/blob/master/docs/networking.md

    Settings to consider (not covered in this guide):

```
--vpc
  Allows you to use a custom VPC or share a VPC
  https://github.com/kubernetes/kops/blob/master/docs/run_in_existing_vpc.md
--master-count
  Spawns more masters in one or more VPCs
  This improves redudancy and reduces downtime during cluster upgrades
--master-zones
  specify zones to run the master in
--node-count
  Increases the total nodes created (default 2)
--master/node-security-groups
  Allows you to specify additional security groups to put the masters and nodes␣
↪in by default
--ssh-access
  By default SSH access is open to the world (0.0.0.0).
  If you are using a private topology, this is not a problem.
  If you are using a public topology make sure your ssh keys are strong and you␣
↪keep sshd up to date on your cluster's nodes.
```

Note: Consider setting a cloud budget for your AWS account in order to make sure you don't accidentally spend more than you wish to.

12. Wait for the cluster to start-up

    Running the 'kops validate cluster' command will tell us what the current state of setup is. If you see "can not get nodes" initially, just be patient as the cluster can't report until a few basic services are up and running.

    Keep running 'kops validate cluster' until you see "Your cluster $NAME is ready" at the end of the output.

    ```
    time until kops validate cluster; do sleep 15 ; done
    ```

    can be used to automate the waiting process.

    If at any point you wish to destroy your cluster after this step, run `kops delete cluster $NAME --yes`

13. Confirm that `kubectl` is connected to your Kubernetes cluster.

    Run:

    ```
    .. code-block:: bash

       kubectl get nodes
    ```

    You should see a list of two nodes, each beginning with `ip`.

    If you want to use kubectl and helm locally (necessary for step #3 in Setting up Helm):

    - run the following on CI host: `kops export kubecfg`
    - copy the contents of `~/.kube/config` to the same place on your local system

    If you wish to put the kube config file in a different location, you will need to

    ```
    export KUBECONFIG=<other kube config location>
    ```

14. Configure ssh bastion (Skip this step if you did not go with the **–topology private** option above!)

Ideally we would simply be passing the –bastion flag into the kops command above. However that flag is not functioning as intended at the moment. https://github.com/kubernetes/kops/issues/2881

Instead we need to follow this guide: https://github.com/kubernetes/kops/blob/master/docs/examples/kops-tests-private-net-bastion-host.md#adding-a-bastion-host-to-our-cluster

At this point there are a few public endpoints left open which need to be addressed

- Bastion ELB security group defaults to access from 0.0.0.0
- API ELB security group defaults to access from 0.0.0.0

15. Enable dynamic storage on your Kubernetes cluster.

    Create a file, `storageclass.yml` on your local computer, and enter this text:

    ```
    kind: StorageClass
    apiVersion: storage.k8s.io/v1
    metadata:
      annotations:
         storageclass.beta.kubernetes.io/is-default-class: "true"
      name: gp2
    provisioner: kubernetes.io/aws-ebs
    parameters:
      type: gp2
    ```

    Next, run this command:

    ```
    kubectl apply -f storageclass.yml
    ```

    This enables dynamic provisioning of disks, allowing us to automatically assign a disk per user when they log in to JupyterHub.

Encryption

There are simple methods for encrypting your Kubernetes cluster. Illustrated here are simple methods for encryption at rest and encryption in transit.

Encryption at Rest

Instead of performing step 13 above. Create the following `storageclass.yml` file on your local computer:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  annotations:
     storageclass.beta.kubernetes.io/is-default-class: "true"
  name: gp2
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  encrypted: "true"
```

The main difference is the addition of the line `encrypted: "true"` and make note that `true` is in double quotes.

Next run these commands:

```
kubectl delete storageclass gp2
kubectl apply -f storageclass.yml
```

Kubernetes will not allow you to modify storageclass gp2 in order to add the `encrypted` flag so you will have to delete it first. This will encrypt any dynamic volumes (such as your notebook)created by Kubernetes, it will not encrypt the storage on the Kubernetes nodes themselves.

Encryption in Transit

In step 9 above, set up the cluster with weave by including the `--networking weave` flag in the `kops create` command above. Then perform the following steps:

1. Verify weave is running:

   ```
   kubectl --namespace kube-system get pods
   ```

   You should see several pods of the form `weave-net-abcde`

2. Create Kubernetes secret with a private password of sufficient strength. A random 128 bytes is used in this example:

   ```
   openssl rand -hex 128 >weave-passwd
   kubectl create secret -n kube-system generic weave-passwd --from-file=./weave-
   ↪passwd
   ```

   It is important that the secret name and its value (taken from the filename) are the same. If they do not match you may get a `ConfigError`

3. Patch Weave with the password:

   ```
   kubectl patch --namespace=kube-system daemonset/weave-net --type json -p
   ↪'[ { "op": "add", "path": "/spec/template/spec/containers/0/env/0",
   ↪"value": { "name": "WEAVE_PASSWORD", "valueFrom": { "secretKeyRef": {
   ↪"key": "weave-passwd", "name": "weave-passwd" } } } } ]'
   ```

   If you want to remove the encryption you can use the following patch:

   ```
   kubectl patch --namespace=kube-system daemonset/weave-net --type json -p
   ↪'[ { "op": "remove", "path": "/spec/template/spec/containers/0/env/0"} ]
   ↪'
   ```

4. Check to see that the pods are restarted. To expedite the process you can delete the old pods.

5. You can verify encryption is turned on with the following command:

   ```
   kubectl exec -n kube-system weave-net-<pod> -c weave -- /home/weave/weave␣
   ↪--local status
   ```

   You should see `encryption: enabled`

   If you really want to insure encryption is working, you can listen on port `6783` of any node. If the traffic looks like gibberish, you know it is on.

Shared Storage

A shared volume is supposed to be mounted to multiple user pods, so we cannot use EBS. As an alternative, there's AWS EFS:

1. amazon-efs
2. *Customizing User Storage*

Congrats. Now that you have your Kubernetes cluster running, it's time to begin *Setup JupyterHub*.

2.1.5 Kubernetes on Amazon Web Services (AWS) with Elastic Container with Kubernetes (EKS)

AWS has released native support for Kubernetes, which is available in selected regions.

This guide uses AWS to set up a cluster. This mirrors the steps found at Getting Started with Amazon EKS with some details filled in that are absent

Procedure

1. Create a IAM Role for EKS Service Role.

   It should have the following policies

   - AmazonEKSClusterPolicy
   - AmazonEKSServicePolicy
   - AmazonEC2ContainerRegistryReadOnly

   (From the user interface, select EKS as the service, then follow the default steps)

2. Create a VPC if you don't already have one.

   This step has a lot of variability so it is left to the user. However, one deployment can be found at Getting Started with Amazon EKS, under *Create your Amazon EKS Cluster VPC*

3. Create a Security Group for the EKS Control Plane to use

   You do not need to set any permissions on this. The steps below will automatically define access control between the EKS Control Plane and the individual nodes

4. Create your EKS cluster (using the user interface)

   Use the IAM Role in step 1 and Security Group defined in step 3. The cluster name is going to be used throughout. We'll use `Z2JHKubernetesCluster` as an example.

5. Install **kubectl** and **aws-iam-authenticator**

   Refer to Getting Started with Amazon EKS on *Configure kubectl for Amazon EKS*

6. Configure *kubeconfig*

   Also see Getting Started with Amazon EKS *Step 2: Configure kubectl for Amazon EKS*

   From the user interface on AWS you can retrieve the `endpoint-url`, `base64-encoded-ca-cert`. `cluster-name` is the name given in step 4. If you are using profiles in your AWS configuration, you can uncomment the `env` block and specify your profile as `aws-profile`.:

```
apiVersion: v1
clusters:
- cluster:
  server: <endpoint-url>
  certificate-authority-data: <base64-encoded-ca-cert>
  name: kubernetes
  contexts:
  - context:
    cluster: kubernetes
    user: aws
    name: aws
    current-context: aws
    kind: Config
```

(continues on next page)

(continued from previous page)

```
    preferences: {}
    users:
    - name: aws
      user:
        exec:
          apiVersion: client.authentication.k8s.io/v1alpha1
          command: aws-iam-authenticator
          args:
          - "token"
          - "-i"
          - "<cluster-name>"
          # env:
          # - name: AWS_PROFILE
          #   value: "<aws-profile>"
```

7. Verify kubectl works

```
kubectl get svc
```

   should return `kubernetes` and `ClusterIP`

8. Create the nodes using CloudFormation

   See Getting Started with Amazon EKS *Step 3: Launch and Configure Amazon EKS Worker Nodes*

   Warning if you are endeavoring to deploy on a private network, the cloudformation template creates a public IP for each worker node though there is no route to get there if you specified only private subnets. Regardless, if you wish to correct this, you can edit the cloudformation template by changing `Resources.NodeLaunchConfig.Properties.AssociatePublicIpAddress` from `'true'` to `'false'`

9. Create a AWS authentication ConfigMap

   This is necessary for the workers to find the master plane.

   See Getting Started with Amazon EKS *Step 3: Launch and Configure Amazon EKS Worker Nodes*

Cluster Autoscaler

If you'd like to do some *optimizations*, you need to deploy Cluster Autoscaler (CA) first.

See https://eksworkshop.com/scaling/deploy_ca/

2.1.6 Kubernetes on Red Hat OpenShift

OpenShift from RedHat is a cluster manager based on Kubernetes.

For setting up JupyterHub on OpenShift, check out the JupyterHub on OpenShift project. It provides an OpenShift template based JupyterHub deployment. Zero to JupyterHub uses helm which is currently usable with OpenShift; yet deploying helm on OpenShift is somewhat complicated (see RedHat's blog post on Getting Started with Helm on OpenShift).

Additional resources about Jupyter on OpenShift

- An excellent series of OpenShift blog posts on Jupyter and OpenShift authored by Red Hat developer, Graham Dumpleton, are available on the OpenShift blog.

2.1.7 Kubernetes on IBM Cloud

This guide shows how to deploy JupyterHub on the IBM Cloud using their Kubernetes service. It should provide you with enough knowledge to create a cluster, deploy your application using a Docker image and use a custom domain to use the deployed app.

Before you begin:

- **Understand the basics of Kubernetes.**
- **Install the IBM Cloud Developer Tools**
  - Install the IBM Cloud CLI.
  - `curl -sL https://ibm.biz/idt-installer | bash`
  - Verify your installation
  - `ibmcloud dev help`
  - Connect to the proper IBM API endpoint for your IBM Cloud location. Example:
  - `ibmcloud api https://api.ng.bluemix.net`
  - Log in to IBM Cloud using your IBMid
  - `ibmcloud login`. Use the `--sso` option to log in using your federated ID.
  - Set up your org and space
  - `ibmcloud target --cf`

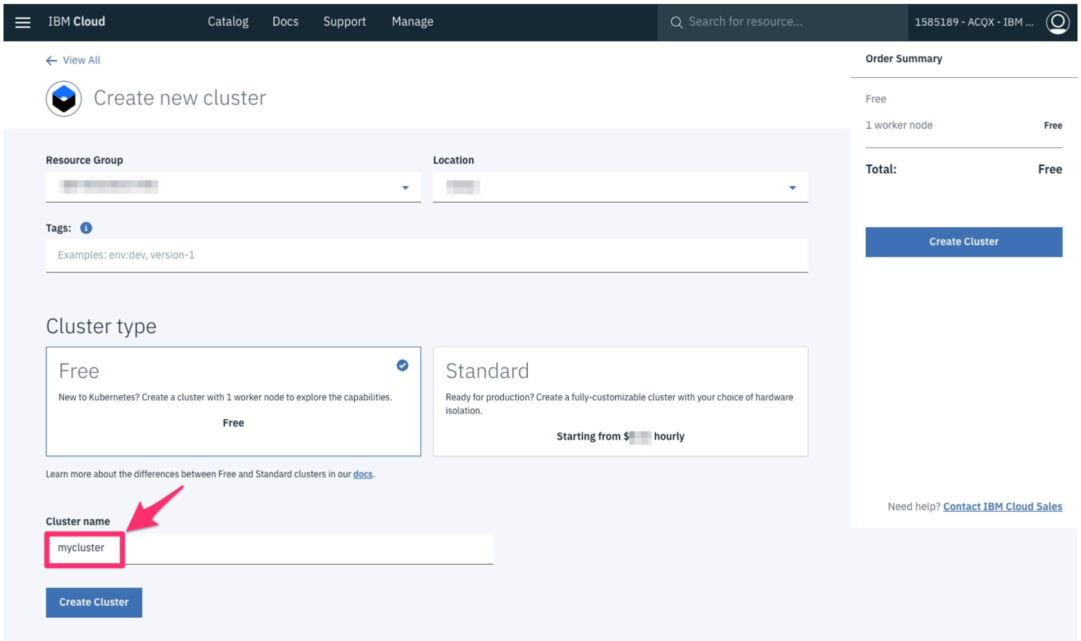
To follow this guide, you can use a **free** cluster. You can also use a **paid** cluster of type **standard** on IBM Cloud.

Procedure:

1. Create a Kubernetes cluster Kubernetes Service delivers powerful tools by combining Docker and Kubernetes technologies, an intuitive user experience, and built-in security and isolation to automate the deployment, operation, scaling, and monitoring of containerized apps in a cluster of computing hosts.

   To set up the Kubernetes cluster:

   1. Create a Kubernetes cluster from the IBM Cloud catalog.
   2. When configuring the new cluster, select the **Cluster type** and click **Create Cluster** to provision a Kubernetes cluster. 2.1 In the case of a free cluster you will see something similar to:



Create new cluster

Resource Group: [redacted]

Location: [redacted]

Tags: Examples: env:dev, version-1

Cluster type

Free ☑
New to Kubernetes? Create a cluster with 1 worker node to explore the capabilities.
Free

Standard
Ready for production? Create a fully-customizable cluster with your choice of hardware isolation.
Starting from $[redacted] hourly

Learn more about the differences between Free and Standard clusters in our docs.

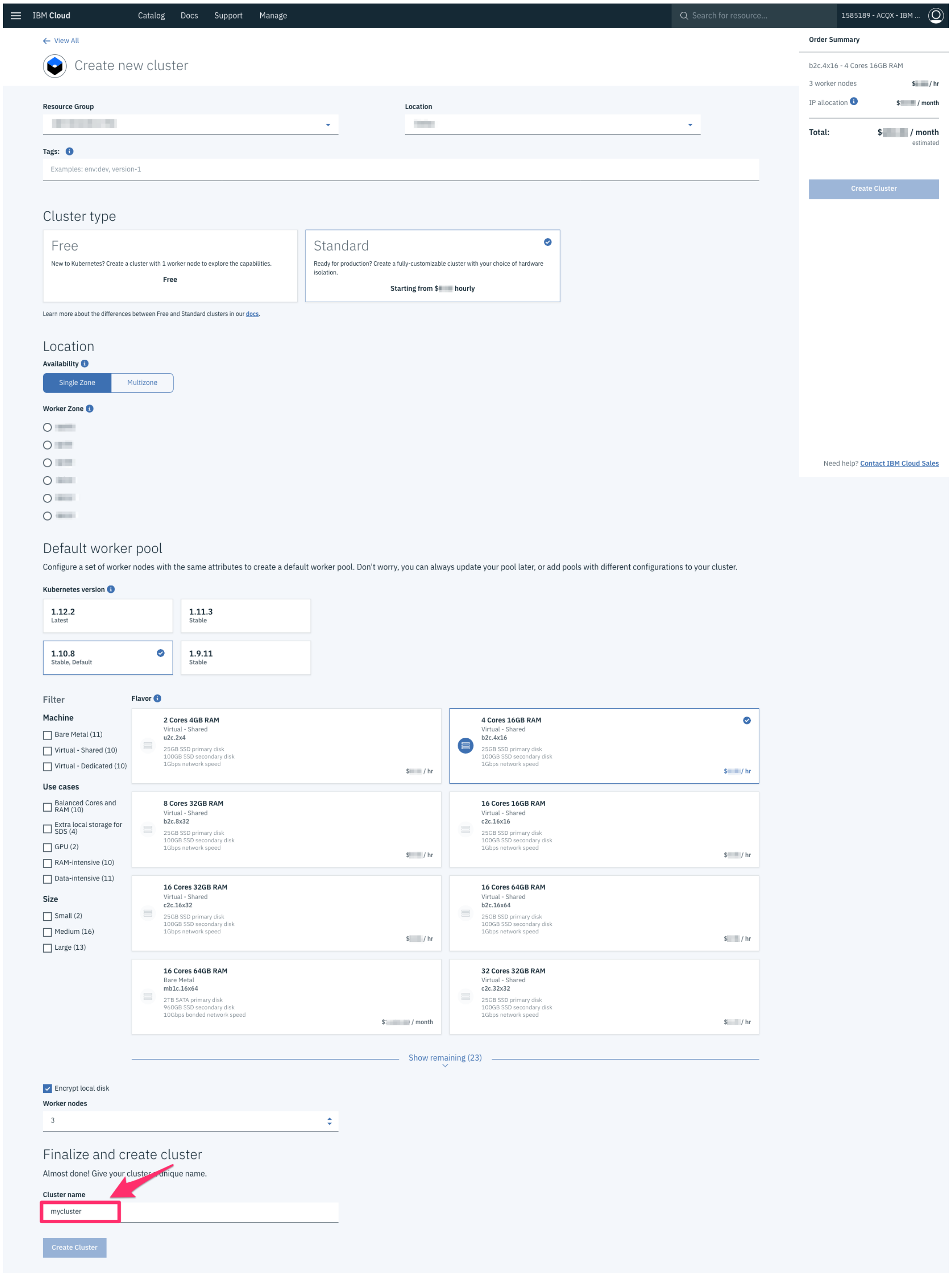
Cluster name: mycluster

Order Summary

Free	
1 worker node	Free
Total:	Free

Need help? Contact IBM Cloud Sales

2.2 In the case of a paid cluster you will see something similar to:



Create new cluster

Resource Group: [redacted]

Location: [redacted]

Tags: Examples: env:dev, version-1

Cluster type

Free
New to Kubernetes? Create a cluster with 1 worker node to explore the capabilities.
Free

Standard ✓
Ready for production? Create a fully-customizable cluster with your choice of hardware isolation.
Starting from $[redacted] hourly

Learn more about the differences between Free and Standard clusters in our docs.

Location

Availability: [Single Zone] Multizone

Worker Zone
○ [redacted]
○ [redacted]
○ [redacted]
○ [redacted]
○ [redacted]
○ [redacted]

Default worker pool

Configure a set of worker nodes with the same attributes to create a default worker pool. Don't worry, you can always update your pool later, or add pools with different configurations to your cluster.

Kubernetes version
- 1.12.2 Latest
- 1.11.3 Stable
- 1.10.8 Stable, Default ✓
- 1.9.11 Stable

Filter

Machine
□ Bare Metal (11)
□ Virtual - Shared (10)
□ Virtual - Dedicated (10)

Use cases
□ Balanced Cores and RAM (10)
□ Extra local storage for SDS (4)
□ GPU (2)
□ RAM-intensive (10)
□ Data-intensive (11)

Size
□ Small (2)
□ Medium (16)
□ Large (13)

Flavor

2 Cores 4GB RAM
Virtual - Shared
u2c.2x4
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

4 Cores 16GB RAM ✓
Virtual - Shared
b2c.4x16
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

8 Cores 32GB RAM
Virtual - Shared
b2c.8x32
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

16 Cores 16GB RAM
Virtual - Shared
c2c.16x16
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

16 Cores 32GB RAM
Virtual - Shared
c2c.16x32
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

16 Cores 64GB RAM
Virtual - Shared
b2c.16x64
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

16 Cores 64GB RAM
Bare Metal
mb1c.16x64
2TB SATA primary disk
960GB SSD secondary disk
10Gbps bonded network speed
$[redacted] / month

32 Cores 32GB RAM
Virtual - Shared
c2c.32x32
25GB SSD primary disk
100GB SSD secondary disk
1Gbps network speed
$[redacted] / hr

Show remaining (23)

☑ Encrypt local disk

Worker nodes: 3

Finalize and create cluster

Almost done! Give your cluster a unique name.

Cluster name: mycluster

Create Cluster

Order Summary

b2c.4x16 - 4 Cores 16GB RAM

3 worker nodes $[redacted] / hr

IP allocation $[redacted] / month

Total: $[redacted] / month
estimated

Create Cluster

Need help? Contact IBM Cloud Sales

3. Check the status of your **Cluster** and **Worker Nodes** and wait for them to be **ready**.

Or, if you prefer, create the cluster using the IBM Cloud CLI tools

2. Configure kubectl

kubectl is a CLI tool to interact with a Kubernetes cluster. In this occasion, you will use it to point forward to the created Kubernetes cluster.

1. Use `ibmcloud login` to log in interactively into the IBM Cloud. Provide the organization (org), loca-

tion and space under which the cluster is created. You can reconfirm the details by running `ibmcloud target` command.

2. **When the cluster is ready, retrieve the cluster configuration by using the cluster's name:**

   ```
   ibmcloud cs cluster-config <clusterName>
   ```

3. **Copy and paste the export command to set the KUBECONFIG environment variable as directed. The command sho**

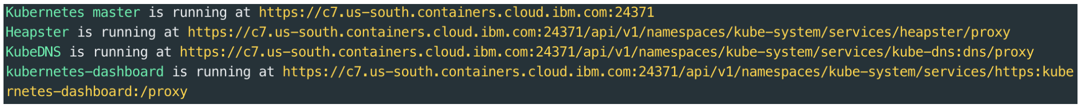
   ```
   export KUBECONFIG=/Users/user/.bluemix/plugins/container-service/clusters/
   ↪JupyterHub/kube-config-***-JupyterHub.yml
   ```

   To verify whether the KUBECONFIG environment variable is set correctly or not, run the following command:

   ```
   echo $KUBECONFIG
   ```

4. **Check that the `kubectl` command is correctly configured**

   ```
   kubectl cluster-info
   ```



```
Kubernetes master is running at https://c7.us-south.containers.cloud.ibm.com:24371
Heapster is running at https://c7.us-south.containers.cloud.ibm.com:24371/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://c7.us-south.containers.cloud.ibm.com:24371/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
kubernetes-dashboard is running at https://c7.us-south.containers.cloud.ibm.com:24371/api/v1/namespaces/kube-system/services/https:kube
rnetes-dashboard:/proxy
```

Hooray! You have your Kubernetes cluster running; it's time to begin *Setup JupyterHub*.

More info and readings:

- https://console.bluemix.net/docs/tutorials/scalable-webapp-kubernetes.html#deploy-a-scalable-web-application-on-kubernetes
- https://github.com/IBM-Cloud/get-started-python

2.1.8 Kubernetes on Digital Ocean

You can create a Kubernetes cluster either through the Digital Ocean website, or using the Digital Ocean command line tools.

This page describes the commands required to setup a Kubernetes cluster using the command line. If you prefer to use the Digital Ocean portal see the Digital Ocean Get Started.

1. Prepare your Digital Ocean shell environment.

   a. **Install command-line tools locally**. You'll need at least v1.13.0.

      You can either follow the `installation instructions <https://github.com/digitalocean/doctl/blob/master/README.md>` or use the commands below:

      ```
      wget https://github.com/digitalocean/doctl/releases/download/v1.13.0/
      ↪doctl-1.13.0-linux-amd64.tar.gz
      tar -xvf doctl-1.13.0-linux-amd64.tar.gz
      sudo mv doctl /usr/bin/
      ```

   b. Create an API token on the Digital Ocean portal. Navigate to API then Generate New Token.

   c. Connect your local CLI with your account:

```
doctl auth init
```

2. Create your cluster. Digital Ocean's use of doctl for kubernetes support is in beta so you'll need to run the following (add it to your `.bashrc` if you want to make this change permanent).

```
export DIGITALOCEAN_ENABLE_BETA=1
doctl k8s cluster create jupyter-kubernetes --region lon1 --version 1.12.1-do.2 --
↪node-pool="name=worker-pool;count=3
```

3. Export your cluster config. You can change the default location from $HOME/.kube by setting the KUBECONFIG environment variable.

```
mkdir -p ~/.kube
doctl k8s cluster kubeconfig show bindertime-k8s > ~/.kube/config
```

4. Create an ssh key to secure your cluster.

```
ssh-keygen -f ssh-key-<CLUSTER-NAME>
```

   It will prompt you to add a password, which you can leave empty if you wish. This will create a public key named `ssh-key-<CLUSTER-NAME>.pub` and a private key named `ssh-key-<CLUSTER-NAME>`. Make sure both go into the folder we created earlier, and keep both of them safe!

   > This command will also print out something to your terminal screen. You don't need to do anything with this text.

5. Check if your cluster is fully functional

```
kubectl get node
```

   The response should list three running nodes and their Kubernetes versions! Each node should have the status of `Ready`, note that this may take a few moments.

Congrats. Now that you have your Kubernetes cluster running, it's time to begin *Setup JupyterHub*.

Note:

- During the process of setting up JupyterHub, you'll be creating some files for configuration purposes. It may be helpful to create a folder for your JuypterHub deployment to keep track of these files.

CHAPTER 3

Setup JupyterHub

This tutorial starts from *Step Zero: your Kubernetes cluster* and describes the steps needed for you to create a complete initial JupyterHub deployment. This will use the JupyterHub Helm chart which provides sensible defaults for an initial deployment.

3.1 Setup JupyterHub

This tutorial starts from *Step Zero: your Kubernetes cluster* and describes the steps needed for you to create a complete initial JupyterHub deployment. This will use the JupyterHub Helm chart which provides sensible defaults for an initial deployment.

3.1.1 Setting up Helm

Helm, the package manager for Kubernetes, is a useful tool for: installing, upgrading and managing applications on a Kubernetes cluster. Helm packages are called *charts*. We will be installing and managing JupyterHub on our Kubernetes cluster using a Helm chart.

Charts are abstractions describing how to install packages onto a Kubernetes cluster. When a chart is deployed, it works as a templating engine to populate multiple `yaml` files for package dependencies with the required variables, and then runs `kubectl apply` to apply the configuration to the resource and install the package.

Helm has two parts: a client (`helm`) and a server (`tiller`). Tiller runs inside of your Kubernetes cluster as a pod in the kube-system namespace. Tiller manages both, the *releases* (installations) and *revisions* (versions) of charts deployed on the cluster. When you run `helm` commands, your local Helm client sends instructions to `tiller` in the cluster that in turn make the requested changes.

Note: These instructions are for Helm 2. Helm 3 includes several major breaking changes and is not yet officially supported, but preliminary instructions are available for testing.

Installation

While several methods to install Helm exists, the simplest way to install Helm is to run Helm's installer script in a terminal:

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
```

Initialization

After installing helm on your machine, initialize Helm on your Kubernetes cluster:

1. Set up a ServiceAccount for use by `tiller`.

   ```
   kubectl --namespace kube-system create serviceaccount tiller
   ```

2. Give the `ServiceAccount` full permissions to manage the cluster.

   > **Note:** If you know your kubernetes cluster does not have RBAC enabled, you **must** skip this step. Most users can ignore this note.

   ```
   kubectl create clusterrolebinding tiller --clusterrole cluster-admin --
   ↪serviceaccount=kube-system:tiller
   ```

   See our RBAC documentation for more information.

3. Initialize `helm` and `tiller`.

   ```
   helm init --service-account tiller --history-max 100 --wait
   ```

   This command only needs to run once per Kubernetes cluster, it will create a `tiller` deployment in the kube-system namespace and setup your local `helm` client. This command installs and configures the `tiller` part of Helm (the whole project, not the CLI) on the remote kubernetes cluster. Later when you want to deploy changes with `helm` (the local CLI), it will talk to `tiller` and tell it what to do. `tiller` then executes these instructions from within the cluster. We limit the history to 100 previous installs as very long histories slow down helm commands a lot.

   > **Note:** If you wish to install `helm` on another computer, you won't need to setup `tiller` again but you still need to initialize `helm`:
   >
   > ```
   > helm init --client-only
   > ```

Secure Helm

Ensure that `tiller` is secure from access inside the cluster:

```
kubectl patch deployment tiller-deploy --namespace=kube-system --type=json --patch='[{
↪"op": "add", "path": "/spec/template/spec/containers/0/command", "value": ["/tiller
↪", "--listen=localhost:44134"]}]'
```

`tiller` s port is exposed in the cluster without authentication and if you probe this port directly (i.e. by bypassing `helm`) then `tiller` s permissions can be exploited. This step forces `tiller` to listen to commands from localhost (i.e. `helm`) *only* so that e.g. other pods inside the cluster cannot ask `tiller` to install a new chart granting them arbitrary, elevated RBAC privileges and exploit them. More details here.

Verify

You can verify that you have the correct version and that it installed properly by running:

```
helm version
```

It should in less then a minute, when `tiller` on the cluster is ready, be able to provide output like below. Make sure you have at least version 2.11.0 and that the client (`helm`) and server version (`tiller`) is matching!

```
Client: &version.Version{SemVer:"v2.11.0", GitCommit:
↪"2e55dbe1fdb5fdb96b75ff144a339489417b146b", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.11.0", GitCommit:
↪"2e55dbe1fdb5fdb96b75ff144a339489417b146b", GitTreeState:"clean"}
```

Next Step

Congratulations, Helm is now set up! Let's continue with *Setting up JupyterHub*!

3.1.2 Setting up JupyterHub

Now that we have a *Kubernetes cluster* and *Helm* setup, we can proceed by using Helm to install JupyterHub and related *Kubernetes resources* using a *Helm chart*.

Prepare configuration file

In this step we will prepare a YAML configuration file that we will refer to as `config.yaml`. It will contain the multiple *Helm values* to be provided to a JupyterHub *Helm chart* developed specifically together with this guide.

Helm charts contains *templates* that with provided values will render to *Kubernetes resources* to be installed in a Kubernetes cluster. This config file will provide the values to be used by our Helm chart.

1. Generate a random hex string representing 32 bytes to use as a security token. Run this command in a terminal and copy the output:

   ```
   openssl rand -hex 32
   ```

2. Create and start editing a file called `config.yaml`. In the code snippet below we start the widely available nano editor, but any editor will do.

   ```
   nano config.yaml
   ```

3. Write the following into the `config.yaml` file but instead of writing `<RANDOM-HEX>` paste the generated hex string you copied in step 1.

   ```
   proxy:
     secretToken: "<RANDOM_HEX>"
   ```

   Note: It is common practice for Helm and Kubernetes YAML files to indent using two spaces.

4. Save the `config.yaml` file. In the nano editor this is done by pressing **CTRL+X** or **CMD+X** followed by a confirmation to save the changes.

Install JupyterHub

1. Make Helm aware of the JupyterHub Helm chart repository so you can install the JupyterHub chart from it without having to use a long URL name.

```
helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/
helm repo update
```

   This should show output like:

```
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "jupyterhub" chart repository
Update Complete.  Happy Helming!
```

2. Now install the chart configured by your `config.yaml` by running this command from the directory that contains your `config.yaml`:

```
# Suggested values: advanced users of Kubernetes and Helm should feel
# free to use different values.
RELEASE=jhub
NAMESPACE=jhub

helm upgrade --install $RELEASE jupyterhub/jupyterhub \
  --namespace $NAMESPACE  \
  --version=0.8.2 \
  --values config.yaml
```

   where:

   - `RELEASE` refers to a Helm release name, an identifier used to differentiate chart installations. You need it when you are changing or deleting the configuration of this chart installation. If your Kubernetes cluster will contain multiple JupyterHubs make sure to differentiate them. You can list your Helm releases with `helm list`.
   - `NAMESPACE` refers to a Kubernetes namespace, an identifier used to group Kubernetes resources, in this case all Kubernetes resources associated with the JupyterHub chart. You'll need the namespace identifier for performing any commands with `kubectl`.

   Note:

   - This step may take a moment, during which time there will be no output to your terminal. JupyterHub is being installed in the background.
   - If you get a `release named <YOUR-RELEASE-NAME> already exists` error, then you should delete the release by running `helm delete --purge <YOUR-RELEASE-NAME>`. Then reinstall by repeating this step. If it persists, also do `kubectl delete namespace <YOUR-NAMESPACE>` and try again.
   - In general, if something goes *wrong* with the install step, delete the Helm release by running `helm delete --purge <YOUR-RELEASE-NAME>` before re-running the install command.

- If you're pulling from a large Docker image you may get a `Error: timed out waiting for the condition` error, add a `--timeout=SOME-LARGE-NUMBER-OF-SECONDS` parameter to the `helm install` command.
- The `--version` parameter corresponds to the *version of the Helm chart*, not the version of JupyterHub. Each version of the JupyterHub Helm chart is paired with a specific version of JupyterHub. E.g., `0.7.0` of the Helm chart runs JupyterHub `0.9.2`. For a list of which JupyterHub version is installed in each version of the Z2JH Helm Chart, see the Helm Chart repository.

3. While Step 2 is running, you can see the pods being created by entering in a different terminal:

```
kubectl get pod --namespace jhub
```

Note: To remain sane we recommend that you enable autocompletion for kubectl and set a default value for the `--namespace` flag:

```
kubectl config set-context $(kubectl config current-context) --namespace $
↪{NAMESPACE:-jhub}
```

4. Wait for the *hub* and *proxy* pod to enter the `Running` state.

5. Find the IP we can use to access the JupyterHub. Run the following command until the `EXTERNAL-IP` of the `proxy-public` service is available like in the example output.

```
kubectl get service --namespace jhub
```

```
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
hub             ClusterIP      10.51.243.14    <none>          8081/TCP         1m
proxy-api       ClusterIP      10.51.247.198   <none>          8001/TCP         1m
proxy-public    LoadBalancer   10.51.248.230   104.196.41.97   80:31916/TCP     1m
```

Note: If the IP for `proxy-public` is too long to fit into the window, you can find the longer version by calling:

```
kubectl describe service proxy-public --namespace jhub
```

7. To use JupyterHub, enter the external IP for the `proxy-public` service in to a browser. JupyterHub is running with a default *dummy* authenticator so entering any username and password combination will let you enter the hub.

Congratulations! Now that you have basic JupyterHub running, you can *extend it* and *optimize it* in many ways to meet your needs.

Some examples of customisations are:

- Configure the login to use the account that makes sense to you (Google, GitHub, etc.).
- Use a suitable pre-built image for the user container or build your own.
- Host it on https://your-domain.com.
- …

3.1.3 Tearing Everything Down

When you are done with your hub, you should delete it so you are no longer paying money for it. The following sections describe how to delete your JupyterHub deployment and associated cloud resources on various cloud providers.

Tearing down your JupyterHub entails:

1. Deleting your Kubernetes namespace, which deletes all objects created and managed by Kubernetes in it.
2. Deleting any cloud resources you've requested from the cloud provider.
3. Running a final check to make sure there aren't any lingering resources that haven't been deleted (e.g., storage volumes in some cloud providers).

For all cloud providers

Delete the helm release

The steps in this section must be performed for all cloud providers first, before doing the cloud provider specific setup.

1. First, delete the Helm release. This deletes all resources that were created by Helm for your JupyterHub deployment.

```
  helm delete <YOUR-HELM-RELEASE-NAME> --purge

``<YOUR-HELM-RELEASE-NAME>`` is the name provided to ``helm upgrade`` when
initially setting up the hub. If you had forgotten what you used, you
can run ``helm list`` to find all the release names in your cluster.
You can also see the ``namespace`` value here that will be used in the next␣
↪step.
```

2. Next, delete the Kubernetes namespace the hub was installed in. This deletes any disks that may have been created to store user's data, and any IP addresses that may have been provisioned.

```
kubectl delete namespace <YOUR-NAMESPACE>
```

Google Cloud Platform

1. Perform the steps in *Delete the helm release*. These cloud provider agnostic steps will delete the Helm release and the Kubernetes namespace. This must be done before proceeding.
2. Delete the Kubernetes cluster. You can list all the clusters you have.

```
gcloud container clusters list
```

   You can then delete the one you want.

```
gcloud container clusters delete <CLUSTER-NAME> --zone=<CLUSTER-ZONE>
```

3. Double check to make sure all the resources are now deleted, since anything you have not deleted will cost you money! You can check the web console (make sure you are in the right project and account) to verify that everything has been deleted.

   At a minimum, check the following under the Hamburger (left top corner) menu:

   1. Compute -> Compute Engine -> Disks
   2. Compute -> Kubernetes Engine -> Clusters

3. Tools -> Container Registry -> Images
4. Networking -> Network Services -> Load Balancing

These might take several minutes to clear up, but they shouldn't have anything related to your JupyterHub cluster after you have deleted the cluster.

Microsoft Azure AKS

1. Perform the steps in *Delete the helm release*. These cloud provider agnostic steps will delete the Helm release and the Kubernetes namespace. This must be done before proceeding.
2. Delete your resource group. You can list your active resource groups with the following command

   ```
   az group list --output table
   ```

   You can then delete the one you want with the following command

   ```
   az group delete --name <YOUR-GROUP-NAME>
   ```

   Be careful to delete the correct Resource Group, as doing so will irreversibly delete all resources within the group!
3. Double check to make sure all the resources are now deleted, since anything you have not deleted will cost you money! You can check the web portal (check the "Resource Groups" page) to verify that everything has been deleted.

   These might take several minutes to clear up, but they shouldn't have anything related to your JupyterHub cluster after you have deleted the resource group.

Amazon Web Services (AWS)

1. Perform the steps in *Delete the helm release*. These cloud provider agnostic steps will delete the Helm release and the Kubernetes namespace. This must be done before proceeding.
2. on CI host:

```
kops delete cluster <CLUSTER-NAME> --yes

# Leave CI host
exit

# Terminate CI host
aws ec2 stop-instances --instance-ids <aws-instance id of CI host>
aws ec2 terminate-instances --instance-ids <aws-instance id of CI host>
```

Note:

- `<CLUSTER NAME>` should be `<SOME NAME>.k8s.local`.
- Stopping the CI host will still incur disk storage and IP address costs, but the host can be restarted at a later date.
- Sometimes AWS fails to delete parts of the stack on a first pass. Be sure to double-check that your stack has in fact been deleted, and re-perform the actions above if needed.

CHAPTER 4

Customization Guide

JupyterHub can be configured and customized to fit a variety of deployment requirements. If you would like to expand JupyterHub, customize its setup, increase the computational resources available for users, or change authentication services, this guide will walk you through the steps. See the *Configuration Reference* for a list of frequently used configurable helm chart fields.

4.1 Customization Guide

JupyterHub can be configured and customized to fit a variety of deployment requirements. If you would like to expand JupyterHub, customize its setup, increase the computational resources available for users, or change authentication services, this guide will walk you through the steps. See the *Configuration Reference* for a list of frequently used configurable helm chart fields.

4.1.1 Customizing your Deployment

The Helm chart used to install your JupyterHub deployment has a lot of options for you to tweak. For a semi-complete reference list of the options, see the *Configuration Reference*.

Applying configuration changes

The general method to modify your Kubernetes deployment is to:

1. Make a change to your `config.yaml`.
2. Run a `helm upgrade`:

```
RELEASE=jhub

helm upgrade $RELEASE jupyterhub/jupyterhub \
  --version=0.8.2 \
  --values config.yaml
```

Note that `helm list` should display `<YOUR_RELEASE_NAME>` if you forgot it.

3. Verify that the *hub* and *proxy* pods entered the `Running` state after the upgrade completed.

   ```
   kubectl get pod --namespace jhub
   ```

For information about the many things you can customize with changes to your Helm chart through values provided to its templates through `config.yaml`, see the *Customization Guide*.

4.1.2 Customizing User Environment

This page contains instructions for common ways to enhance the user experience. For a list of all the configurable Helm chart options, see the *Configuration Reference*.

The *user environment* is the set of software packages, environment variables, and various files that are present when the user logs into JupyterHub. The user may also see different tools that provide interfaces to perform specialized tasks, such as JupyterLab, RStudio, RISE and others.

A *Docker image* built from a *Dockerfile* will lay the foundation for the environment that you will provide for the users. The image will for example determine what Linux software (curl, vim . . .), programming languages (Julia, Python, R, . . .) and development environments (JupyterLab, RStudio, . . .) are made available for use.

To get started customizing the user environment, see the topics below.

Choose and use an existing Docker image

Project Jupyter maintains the jupyter/docker-stacks repository, which contains ready to use Docker images. Each image includes a set of commonly used science and data science libraries and tools. They also provide excellent documentation on how to choose a suitable image.

If you wish to use another image from jupyter/docker-stacks than the base-notebook used by default, such as the datascience-notebook image containing useful tools and libraries for datascience, complete these steps:

1. Modify your `config.yaml` file to specify the image. For example:

   ```
   singleuser:
     image:
       # Get the latest image tag at:
       # https://hub.docker.com/r/jupyter/datascience-notebook/tags/
       # Inspect the Dockerfile at:
       # https://github.com/jupyter/docker-stacks/tree/master/datascience-notebook/
   ↪Dockerfile
       name: jupyter/datascience-notebook
       tag: 177037d09156
   ```

   Note: Container image names cannot be longer than 63 characters.

   Always use an explicit `tag`, such as a specific commit. Avoid using `latest` as it might cause a several minute delay, confusion, or failures for users when a new version of the image is released.

2. Apply the changes by following the directions listed in *apply the changes*.

   Note: If you have configured *prePuller.hook.enabled*, all the nodes in your cluster will pull the image before the hub is upgraded to let users use the image. The image pulling may take several minutes to complete, depending on the size of the image.

3. Restart your server from JupyterHub control panel if you are already logged in.

Note: If you'd like users to select an environment from **multiple docker images**, see *Using multiple profiles to let users select their environment*.

Use JupyterLab by default

JupyterLab is a new user interface for Jupyter about to replace the classic user interface (UI). While users already can interchange `/tree` and `/lab` in the URL to switch between the classic UI and JupyterLab, they will default to use the classic UI.

To let users use JupyterLab by default, add the following entries to your *config.yaml*:

```yaml
singleuser:
  defaultUrl: "/lab"

hub:
  extraConfig:
    jupyterlab: |
      c.Spawner.cmd = ['jupyter-labhub']
```

Note: All images in the jupyter/docker-stacks repository come pre-installed with JupyterLab and the JupyterLab-Hub extension required for this configuration to work.

Customize an existing Docker image

If you are missing something in the image that you would like all users to have, we recommend that you build a new image on top of an existing Docker image from jupyter/docker-stacks.

Below is an example *Dockerfile* building on top of the *minimal-notebook* image. This file can be built to a *docker image*, and pushed to a *image registry*, and finally configured in *config.yaml* to be used by the Helm chart.

```dockerfile
FROM jupyter/minimal-notebook:177037d09156
# Get the latest image tag at:
# https://hub.docker.com/r/jupyter/minimal-notebook/tags/
# Inspect the Dockerfile at:
# https://github.com/jupyter/docker-stacks/tree/master/minimal-notebook/Dockerfile

# install additional package...
RUN pip install --no-cache-dir astropy
```

Set environment variables

One way to affect your user's environment is by setting *environment variables*. While you can set them up in your Docker image if you build it yourself, it is often easier to configure your Helm chart through values provided in your *config.yaml*.

To set this up, edit your *config.yaml* and *apply the changes*. For example, this code snippet will set the environment variable `EDITOR` to the value `vim`:

```
singleuser:
  extraEnv:
    EDITOR: "vim"
```

You can set any number of static environment variables in the *config.yaml* file.

Users can read the environment variables in their code in various ways. In Python, for example, the following code reads an environment variable's value:

```
import os
my_value = os.environ["MY_ENVIRONMENT_VARIABLE"]
```

About user storage and adding files to it

It is important to understand the basics of how user storage is set up. By default, each user will get 10GB of space on a harddrive that will persist in between restarts of their server. This harddrive will be mounted to their home directory. In practice this means that everything a user writes to the home directory (`/home/jovyan`) will remain, and everything else will be reset in between server restarts.

A server can be shut down by *culling*. By default, JupyterHub's culling service is configured to cull a users server that has been inactive for one hour. Note that JupyterLab will autosave files, and as long as the file was within the users home directory no work is lost.

Note: In Kubernetes, a *PersistantVolume* (PV) represents the harddrive. KubeSpawner will create a PersistantVolumeClaim that requests a PV from the cloud. By default, deleting the PVC will cause the cloud to delete the PV.

Docker image's $HOME directory will be hidden from the user. To make these contents visible to the user, you must pre-populate the user's filesystem. To do so, you would include commands in the `config.yaml` that would be run each time a user starts their server. The following pattern can be used in *config.yaml*:

```
singleuser:
  lifecycleHooks:
    postStart:
      exec:
        command: ["cp", "-a", "src", "target"]
```

Each element of the command needs to be a separate item in the list. Note that this command will be run from the `$HOME` location of the user's running container, meaning that commands that place files relative to `./` will result in users seeing those files in their home directory. You can use commands like `wget` to place files where you like.

A simple way to populate the notebook user's home directory is to add the required files to the container's `/tmp` directory and then copy them to `/home/jovyan` using a `postStart` hook. This example shows the use of multiple commands.

```
singleuser:
 lifecycleHooks:
     postStart:
       exec:
         command:
           - "sh"
           - "-c"
           - >
             cp -r /tmp/foo /home/jovyan;
             cp -r /tmp/bar /home/jovyan
```

Keep in mind that commands will be run **each time** a user starts their server. For this reason, we recommend using `nbgitpuller` to synchronize your user folders with a git repository.

Using `nbgitpuller` to synchronize a folder

We recommend using the tool nbgitpuller to synchronize a folder in your user's filesystem with a `git` repository whenever a user starts their server. This synchronization can also be triggered by letting a user visit a link like `https://your-domain.com/hub/user-redirect/git-pull?repo=https://github.com/data-8/materials-fa18` (e.g., as alternative start url).

To use `nbgitpuller`, first make sure that you install it in your Docker image. Once this is done, you'll have access to the `nbgitpuller` CLI from within JupyterHub. You can run it with a `postStart` hook with the following configuration

```
singleuser:
  lifecycleHooks:
    postStart:
      exec:
        command: ["gitpuller", "https://github.com/data-8/materials-fa17", "master",
↪"materials-fa"]
```

This will synchronize the master branch of the repository to a folder called `$HOME/materials-fa` each time a user logs in. See the nbgitpuller documentation for more information on using this tool.

> **Warning:** `nbgitpuller` will attempt to automatically resolve merge conflicts if your user's repository has changed since the last sync. You should familiarize yourself with the nbgitpuller merging behavior prior to using the tool in production.

Allow users to create their own `conda` environments for notebooks

Sometimes you want users to be able to create their own `conda` environments. By default, any environments created in a JupyterHub session will not persist across sessions. To resolve this, take the following steps:

1. Ensure the `nb_conda_kernels` package is installed in the root environment (e.g., see r2d-custom-image)
2. Configure Anaconda to install user environments to a folder within `$HOME`.

   Create a file called `.condarc` in the home folder for all users, and make sure that the following lines are inside:

   ```
   envs_dirs:
     - /home/jovyan/my-conda-envs/
   ```

   The text above will cause Anaconda to install new environments to this folder, which will persist across sessions.

These environments are supposed to be used in notebooks, so a typical use case:

1. Create one with at least a kernel, e.g. for Python it's `conda create -n myenv ipykernel scipy`
2. Now this env should be available in the list of kernels

Using multiple profiles to let users select their environment

You can create configurations for multiple user environments, and let users select from them once they log in to your JupyterHub. This is done by creating multiple **profiles**, each of which is attached to a set of configuration options that override your JupyterHub's default configuration (specified in your Helm Chart). This can be used to let users choose among many Docker images, to select the hardware on which they want their jobs to run, or to configure default interfaces such as Jupyter Lab vs. RStudio.

Each configuration is a set of options for Kubespawner, which defines how Kubernetes should launch a new user server pod. Any configuration options passed to the `profileList` configuration will overwrite the defaults in Kubespawner (or any configuration you've added elsewhere in your helm chart).

Profiles are stored under `singleuser.profileList`, and are defined as a list of profiles with specific configuration options each. Here's an example:

```
singleuser:
  profileList:
    - display_name: "Name to be displayed to users"
      description: "Longer description for users."
      # Configuration unique to this profile
      kubespawner_override:
        your_config: "Your value"
      # Defines the default profile - only use for one profile
      default: true
```

The above configuration will show a screen with information about this profile displayed when users start a new server.

Here's an example with four profiles that lets users select the environment they wish to use.

```
singleuser:
  # Defines the default image
  image:
    name: jupyter/minimal-notebook
    tag: 2343e33dec46
  profileList:
    - display_name: "Minimal environment"
      description: "To avoid too much bells and whistles: Python."
      default: true
    - display_name: "Datascience environment"
      description: "If you want the additional bells and whistles: Python, R, and␣
↪Julia."
      kubespawner_override:
        image: jupyter/datascience-notebook:2343e33dec46
    - display_name: "Spark environment"
      description: "The Jupyter Stacks spark image!"
      kubespawner_override:
        image: jupyter/all-spark-notebook:2343e33dec46
    - display_name: "Learning Data Science"
      description: "Datascience Environment with Sample Notebooks"
      kubespawner_override:
        image: jupyter/datascience-notebook:2343e33dec46
        lifecycle_hooks:
          postStart:
            exec:
              command:
                - "sh"
                - "-c"
                - >
                  gitpuller https://github.com/data-8/materials-fa17 master materials-
↪fa;
```

This allows users to select from three profiles, each with their own environment (defined by each Docker image in the configuration above).

The "Learning Data Science" environment in the above example overrides the postStart lifecycle hook. Note that when using `kubespawner_override` the values must be in the format that comply with the KubeSpawner configuration. For instance, when overriding the lifecycle hooks in `kubespawner_override`, the configuration is for `lifecycle_hooks` (snake_case) rather than `lifecycleHooks` (camelCase) which is how it is used directly under the `singleuser` configuration section. A further explanation for this can be found in this github issue.

Note: You can also **control the HTML used for the profile selection page** by using the Kubespawner `profile_form_template` configuration. See the Kubespawner configuration reference for more information.

4.1.3 Customizing User Resources

Note: For a list of all the Helm chart options you can configure, see the *Configuration Reference*.

User resources include the CPU, RAM, and Storage which JupyterHub provides to users. Most of these can be controlled via modifications to the Helm chart. For information on deploying your modifications to the JupyterHub deployment, see *Applying configuration changes*.

Since JupyterHub can serve many different types of users, JupyterHub managers and administrators must be able to flexibly **allocate user resources**, like memory or compute. For example, the Hub may be serving power users with large resource requirements as well as beginning users with more basic resource needs. The ability to customize the Hub's resources to satisfy both user groups improves the user experience for all Hub users.

Set user memory and CPU guarantees / limits

Each user on your JupyterHub gets a slice of memory and CPU to use. There are two ways to specify how much users get to use: resource *guarantees* and resource *limits*.

A resource *guarantee* means that all users will have *at least* this resource available at all times, but they may be given more resources if they're available. For example, if users are *guaranteed* 1G of RAM, users can technically use more than 1G of RAM if these resources aren't being used by other users.

A resource *limit* sets a hard limit on the resources available. In the example above, if there were a 1G memory limit, it would mean that users could use no more than 1G of RAM, no matter what other resources are being used on the machines.

By default, each user is *guaranteed* 1G of RAM. All users have *at least* 1G, but they can technically use more if it is available. You can easily change the amount of these resources, and whether they are a *guarantee* or a *limit*, by changing your `config.yaml` file. This is done with the following structure.

```yaml
singleuser:
  memory:
    limit: 1G
    guarantee: 1G
```

This sets a memory limit and guarantee of 1G. Kubernetes will make sure that each user will always have access to 1G of RAM, and requests for more RAM will fail (your kernel will usually die). You can set the limit to be higher than the guarantee to allow some users to use larger amounts of RAM for a very short-term time (e.g. when running a single, short-lived function that consumes a lot of memory).

Similarly, you can limit CPU as follows:

```
singleuser:
  cpu:
    limit: .5
    guarantee: .5
```

This would limit your users to a maximum of .5 of a CPU (so 1/2 of a CPU core), as well as guarantee them that same amount.

Note: Remember to *apply the change* after changing your `config.yaml` file!

Set user GPU guarantees / limits

It is possible to allocate GPUs to your user. This is useful for heavier workloads, such as deep learning, that can take advantage of GPUs.

For example, to create a profile that allocates one NVIDIA GPU:

```
singleuser:
 profileList:
   - display_name: "GPU Server"
     description: "Spawns a notebook server with access to a GPU"
     kubespawner_override:
       extra_resource_limits:
         nvidia.com/gpu: "1"
```

This assumes that at least one of your Kubernetes nodes has compatible GPUs attached. The method for doing this differs according to your infrastructure provider. Here are a few links to help you get started:

- Google Kubernetes Engine (GKE)
- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)

You will also need to deploy the k8s-device-plugin following the instructions here.

To check that your GPUs are schedulable by Kubernetes, you can run the following command:

```
kubectl get nodes -o=custom-columns=NAME:.metadata.name,GPUs:.status.
↪capacity.'nvidia\.com/gpu'
```

Modifying user shared memory size

It is also beneficial to increase the shared memory (SHM) allocation on pods running workloads like deep learning. This is required for functions like PyTorch's DataLoader to run properly.

The following configuration will increase the SHM allocation by mounting a `tmpfs` (ramdisk) at `/dev/shm`, replacing the default 64MB allocation.

```
singleuser:
 storage:
   extraVolumes:
     - name: shm-volume
       emptyDir:
         medium: Memory
```

(continues on next page)

(continued from previous page)

```
    extraVolumeMounts:
      - name: shm-volume
        mountPath: /dev/shm
```

The volume `shm-volume` will be created when the user's pod is created, and destroyed after the pod is destroyed.

Some important notes regarding SHM allocation:

- SHM usage by the pod will count towards its memory limit
- When the memory limit is exceeded, the pod will be evicted

Modifying user storage type and size

See the *Customizing User Storage* for information on how to modify the type and size of storage that your users have access to.

Expanding and contracting the size of your cluster

You can easily scale up or down your cluster's size to meet usage demand or to save cost when the cluster is not being used. This is particularly useful when you have predictable spikes in usage. For example, if you are organizing and running a workshop, resizing a cluster gives you a way to save cost and prepare JupyterHub before the event. For example:

- **One week before the workshop:** You can create the cluster, set everything up, and then resize the cluster to zero nodes to save cost.
- **On the day of the workshop:** You can scale the cluster up to a suitable size for the workshop. This workflow also helps you avoid scrambling on the workshop day to set up the cluster and JupyterHub.
- **After the workshop:** The cluster can be deleted.

The following sections describe how to resize the cluster on various cloud platforms.

Google Cloud Platform

Use the `resize` command and provide a new cluster size (i.e. number of nodes) as a command line option `--size`:

```
gcloud container clusters resize \
    <YOUR-CLUSTER-NAME> \
    --size <NEW-SIZE> \
    --zone <YOUR-CLUSTER-ZONE>
```

To display the cluster's name, zone, or current size, use the command:

```
gcloud container clusters list
```

After resizing the cluster, it may take a couple of minutes for the new cluster size to be reported back as the service is adding or removing nodes. You can find the true count of currently 'ready' nodes using `kubectl get node` to report the current `Ready/NotReady` status of all nodes in the cluster.

Microsoft Azure Platform

Use the `scale` command and provide a new cluster size (i.e. number of nodes) as a command line option `--node-count`:

```
az aks scale \
    --name <YOUR-CLUSTER-NAME> \
    --node-count <NEW-SIZE> \
    --resource-group <YOUR-RESOURCE-GROUP>
```

To display the details of the cluster, use the command:

```
az aks show --name <YOUR-CLUSTER-NAME> --resource-group <YOUR-RESOURCE-GROUP>
```

It may take some time for the new cluster nodes to be ready. You can use `kubectl get node` to report the current `Ready/NotReady` status of all nodes in the cluster.

4.1.4 Customizing User Storage

For the purposes of this guide, we'll describe "storage" as a "volume" - a location on a disk where a user's data resides.

Kubernetes handles the creation and allocation of persistent volumes, under-the-hood it uses the cloud provider's API to issue the proper commands. To that extent most of our discussion around volumes will describe Kubernetes objects.

JupyterHub uses Kubernetes to manage user storage. There are two primary Kubernetes objects involved in allocating storage to pods:

- A `PersistentVolumeClaim` (`PVC`) specifies what kind of storage is required. Its configuration is specified in your `config.yaml` file.
- A `PersistentVolume` (`PV`) is the actual volume where the user's data resides. It is created by Kubernetes using details in a `PVC`.

As Kubernetes objects, they can be queried with the standard `kubectl` commands (e.g., `kubectl --namespace=<your-namespace> get pvc`)

In JupyterHub, each user gets their own `PersistentVolumeClaim` object, representing the data attached to their account. When a new user starts their JupyterHub server, a `PersistentVolumeClaim` is created for that user. This claim tells Kubernetes what kind of storage (e.g., ssd vs. hd) as well as how much storage is needed. Kubernetes checks to see whether a `PersistentVolume` object for that user exists (since this is a new user, none will exist). If no `PV` object exists, then Kubernetes will use the `PVC` to create a new `PV` object for the user.

Now that a `PV` exists for the user, Kubernetes next must attach (or "mount") that `PV` to the user's pod (which runs user code). Once this is accomplished, the user will have access to their `PV` within JupyterHub. Note that this all happens under-the-hood and automatically when a user logs in.

`PersistentVolumeClaims` and `PersistentVolumes` are not deleted unless the `PersistentVolumeClaim` is explicitly deleted by the JupyterHub administrator. When a user shuts down their server, their user pod is deleted and their volume is detached from the pod, *but the `PVC` and `PV` objects still exist*. In the future, when the user logs back in, JupyterHub will detect that the user has a pre-existing `PVC` and will simply attach it to their new pod, rather than creating a new `PVC`.

How can this process break down?

When Kubernetes uses the `PVC` to create a new user `PV`, it is sending a command to the underlying API of whatever cloud provider Kubernetes is running on. Occasionally, the request for a specific `PV` might fail - for example, if your account has reached the limit in the amount of disk space available.

Another common issue is limits on the number of volumes that may be simultaneously attached to a node in your cluster. Check your cloud provider for details on the limits of storage resources you request.

Note: Some cloud providers have a limited number of disks that can be attached to each node. Since JupyterHub allocates one disk per user for persistent storage, this limits the number of users that can be running in a node at any point of time. If you need users to have persistent storage, and you end up hitting this limit, you must use *more* nodes in order to accommodate the disk for each user. In this case, we recommend allocating *fewer* resources per node (e.g. RAM) since you'll have fewer users packed onto a single node.

Configuration

Most configuration for storage is done at the cluster level and is not unique to JupyterHub. However, some bits are, and we will demonstrate here how to configure those.

Note that new `PVCs` for pre-existing users will **not** be created unless the old ones are destroyed. If you update your users' `PVC` config via `config.yaml`, then any **new** users will have the new `PVC` created for them, but **old** users will not. To force an upgrade of the storage type for old users, you will need to manually delete their `PVC` (e.g. `kubectl --namespace=<your-namespace> delete pvc <pvc-name>`). **This will delete all of the user's data** so we recommend backing up their filesystem first if you want to retain their data.

After you delete the user's `PVC`, upon their next log-in a new `PVC` will be created for them according to your updated `PVC` specification.

Type of storage provisioned

A StorageClass object is used to determine what kind of `PersistentVolumes` are provisioned for your users. Most popular cloud providers have a `StorageClass` marked as default. You can find out your default `StorageClass` by doing:

```
kubectl get storageclass
```

and looking for the object with `(default)` next to its name.

To change the kind of `PersistentVolumes` provisioned for your users,

1. Create a new `StorageClass` object following the kubernetes documentation
2. Specify the name of the `StorageClass` you just created in `config.yaml`

```
singleuser:
  storage:
    dynamic:
      storageClass: <storageclass-name>
```

3. Do a `helm upgrade`

Note that this will only affect new users who are logging in. We recommend you do this before users start heavily using your cluster.

We will provide examples for popular cloud providers here, but will generally defer to the Kubernetes documentation.

Google Cloud

On Google Cloud, the default `StorageClass` will provision Standard Google Persistent Disks. These run on Hard Disks. For more performance, you may want to use SSDs. To use SSDs, you can create a new `StorageClass` by first putting the following `yaml` into a new file. We recommend a descriptive name such as `storageclass.yaml`, which we'll use below:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: jupyterhub-user-ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zones: <your-cluster-zone>
```

Replace `<your-cluster-zone>` with the Zone in which you created your cluster (you can find this with `gcloud container clusters list`).

Next, create this object by running `kubectl apply -f storageclass.yaml` from the commandline. The Kubernetes Docs have more information on what the various fields mean. The most important field is `parameters.type`, which specifies the type of storage you wish to use. The two options are:

- `pd-ssd` makes `StorageClass` provision SSDs.
- `pd-standard` will provision non-SSD disks.

Once you have created this `StorageClass`, you can configure your JupyterHub's `PVC` template with the following in your `config.yaml`:

```
singleuser:
  storage:
    dynamic:
      storageClass: jupyterhub-user-ssd
```

Note that for `storageClass:` we use the name that we specified above in `metadata.name`.

Size of storage provisioned

You can set the size of storage requested by JupyterHub in the `PVC` in your `config.yaml`.

```
storage:
  capacity: 2Gi
```

This will request a `2Gi` volume per user. The default requests a `10Gi` volume per user.

We recommend you use the IEC Prefixes (Ki, Mi, Gi, etc) for specifying how much storage you want. `2Gi` (IEC Prefix) is `(2 * 1024 * 1024 * 1024)` bytes, while `2G` (SI Prefix) is `(2 * 1000 * 1000 * 1000)` bytes.

Turn off per-user persistent storage

If you do not wish for users to have any persistent storage, it can be turned off. Edit the `config.yaml` file and set the storage type to `none`:

```
singleuser:
  storage:
    type: none
```

Next *apply the changes*.

After the changes are applied, new users will no longer be allocated a persistent `$HOME` directory. Any currently running users will still have access to their storage until their server is restarted. You might have to manually delete current users' `PVCs` with `kubectl` to reclaim any cloud disks that might have allocated. You can get a current list of `PVCs` with:

```
kubectl --namespace=<your-namespace> get pvc
```

You can then delete the `PVCs` you do not want with:

```
kubectl --namespace=<your-namespace> delete pvc <pvc-name>
```

Remember that deleting someone's `PVCs` will delete all their data, so do so with caution!

Additional storage volumes

If you already have a `PersistentVolume` and `PersistentVolumeClaim` created outside of JupyterHub you can mount them inside the user pods. For example, if you have a shared `PersistentVolumeClaim` called `jupyterhub-shared-volume` you could mount it as `/home/shared` in all user pods:

```
singleuser:
  storage:
    extraVolumes:
      - name: jupyterhub-shared
        persistentVolumeClaim:
          claimName: jupyterhub-shared-volume
    extraVolumeMounts:
      - name: jupyterhub-shared
        mountPath: /home/shared
```

Note that if you want to mount a volume into multiple pods the volume must support a suitable access mode.

4.1.5 Customizing User Management

This section describes management of users and their permissions on JupyterHub.

Culling user pods

JupyterHub will automatically delete any user pods that have no activity for a period of time. This helps free up computational resources and keeps costs down if you are using an autoscaling cluster. When these users navigate back to your JupyterHub, they will have to start their server again, and the state of their previous session (variables they've created, any in-memory data, etc) will be lost. This is known as *culling*.

Note: In JupyterHub, "inactivity" is defined as no response from the user's browser. JupyterHub constantly pings the user's JupyterHub browser session to check whether it is open. This means that leaving the computer running with the JupyterHub window open will **not** be treated as inactivity.

To disable culling, put the following into `config.yaml`:

```
cull:
  enabled: false
```

By default, JupyterHub will run the culling process every ten minutes and will cull any user pods that have been inactive for more than one hour. You can configure this behavior in your `config.yaml` file with the following fields:

```
cull:
  timeout: <max-idle-seconds-before-user-pod-is-deleted>
  every: <number-of-seconds-this-check-is-done>
```

Note: While JupyterHub automatically runs the culling process, it is not a replacement for keeping an eye on your cluster to make sure resources are being used as expected.

Admin Users

JupyterHub has the concept of admin users who have special rights. They can start / stop other user's servers, and optionally access user's notebooks. They will see a new **Admin** button in their Control Panel which will take them to an **Admin Panel** where they can perform all these actions.

You can specify a list of admin users in your `config.yaml`:

```
auth:
  admin:
    users:
      - adminuser1
      - adminuser2
```

By default, admins can access user's notebooks. If you wish to disable this, use this in your `config.yaml`:

```
auth:
  admin:
    access: false
```

Authenticating Users

For information on authenticating users in JupyterHub, see *the Authentication guide*.

CHAPTER 5

Administrator Guide

This section provides information on managing and maintaining a staging or production deployment of JupyterHub. It has considerations for managing cloud-based deployments and tips for maintaining your deployment.

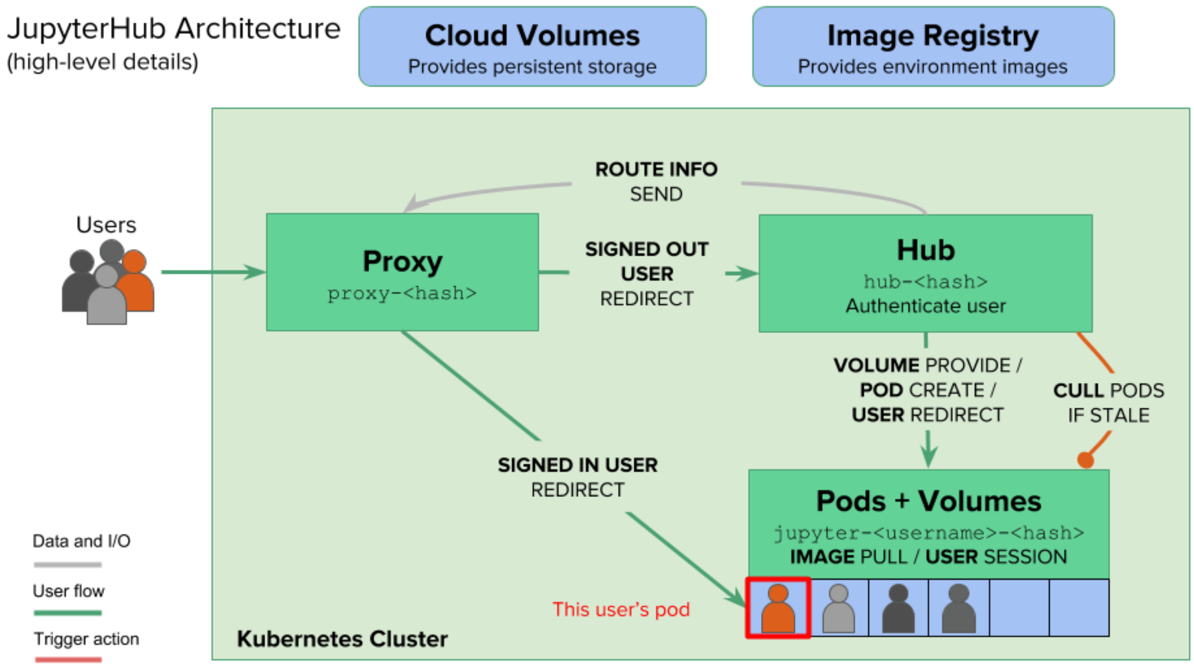
5.1 Administrator Guide

This section provides information on managing and maintaining a staging or production deployment of JupyterHub. It has considerations for managing cloud-based deployments and tips for maintaining your deployment.

5.1.1 The JupyterHub Architecture

The JupyterHub Helm Chart manages resources in the cloud using Kubernetes. There are several moving pieces that, together, handle authenticating users, pulling a Docker image specified by the administrator, generating the user pods in which users will work, and connecting users with those pods.

The following diagram gives a high-level overview of the many pieces of JupyterHub, and how they fit together in this process:



5.1.2 Debugging

Sometimes your JupyterHub deployment doesn't behave the way you'd expect. This section provides some tips on debugging and fixing some common problems.

Debugging commands

In order to debug your JupyterHub deployment, you need to be able to inspect the state of the resources being used. The following are a few common commands for debugging.

Real world scenario: Let's say you've got a JupyterHub deployed, and a user tells you that they are experiencing strange behavior. Let's take a look at our deployment to figure out what is going on.

Note: For our real world scenario, we'll assume that our Kubernetes namespace is called `jhub`. Your namespace may be called something different

`kubectl get pod`

To list all pods in your Kubernetes deployment:

```
kubectl --namespace=jhub get pod
```

This will output a list of all pods being used in the deployment.

Real world scenario: In our case, we see two pods for the JupyterHub infrastructure (`hub` and `proxy`) as well as one user pod that was created when somebody logged in to the JupyterHub.

Here's an example of the output:

```
$ kubectl --namespace=jhub get pod
NAME                                    READY     STATUS         RESTARTS   AGE
hub-3311438805-xnfvp      1/1       Running        0          2m
jupyter-choldgraf                       0/1       ErrImagePull   0          25s
proxy-1227971824-mn2wd    1/1       Running        0          5h
```

Here we can see the two JupyterHub pods, as well as a single user pod. Note that all user pods will begin with `jupyter-`.

In particular, keep an eye on the `STATUS` column. If a given pod contains something other than `Running`, then something may be wrong.

In this case, we can see that our user's pod is in the `ErrImagePull` state. This generally means that there's something wrong with the Docker image that is defined in `singleuser` in our helm chart config. Let's dig further. . .

`kubectl describe pod`

To see more detail about the state of a specific pod, use the following command:

```
kubectl --namespace=jhub describe pod <POD_NAME>
```

This will output several pieces of information, including configuration and settings for the pod. The final section you'll see is a list of recent events. These can be particularly informative, as often an error will show up in this section.

Real world scenario: In our case, one of the lines in the events page displays an error:

```
$ kubectl --namespace=jhub describe pod jupyter-choldgraf
...
2m            52s             4       kubelet, gke-jhubtest-default-pool-52c36683-
↪jv6r         spec.containers{notebook}       Warning         Failed            Failed␣
↪to pull image "jupyter/scipy-notebook:v0.4": rpc error: code = 2 desc = Error␣
↪response from daemon: {"message":"manifest for jupyter/scipy-notebook:v0.4 not found
↪"}
...
```

It seems there is indeed something wrong with the Docker image. Let's confirm this by getting another view on the events that have transpired in the pod.

`kubectl logs`

If you only want to see the latest logs for a pod, use the following command:

```
kubectl --namespace=jhub logs <POD_NAME>
```

This will show you the logs from the pod, which often contain useful information about what is going wrong. Parse these logs to see if something is generating an error.

Real world scenario: In our case, we get this line back:

```
$ kubectl --namespace=jhub logs jupyter-choldgraf
Error from server (BadRequest): container "notebook" in pod "jupyter-choldgraf" is␣
↪waiting to start: trying and failing to pull image
```

Now we are sure that something is wrong with our Dockerfile. Let's check our `config.yaml` file for the section where we specify the user's Docker image. Here we see our problem:

```
singleuser:
  image:
    name: jupyter/scipy-notebook
```

We haven't specified a `tag` for our Docker image! Not specifying a tag will cause it to default to `v0.4`, which isn't what we want and is causing the pod to fail.

To fix this, let's add a tag to our `config.yaml` file:

```
singleuser:
  image:
    name: jupyter/scipy-notebook
    tag: ae885c0a6226
```

Then run a helm upgrade:

```
helm upgrade jhub jupyterhub/jupyterhub --version=v0.6 -f config.yaml
```

where `jhub` is the helm release name (substitute the release name that you chose during setup).

Note: Depending on the size of the Docker image, this may take a while to complete.

Right after you run this command, let's once again list the pods in our deployment:

```
$ kubectl --namespace=jhub get pod
NAME                                READY     STATUS              RESTARTS   AGE
hub-2653507799-r7wf8     0/1       ContainerCreating   0          31s
hub-3311438805-xnfvp     1/1       Terminating         0          14m
jupyter-choldgraf                   0/1       ImagePullBackOff    0          12m
proxy-deployment-1227971824-mn2wd   1/1       Running             0          5h
```

Here we can see one `hub` pod being destroyed, and another (based on the upgraded helm chart) being created. We also see our broken user pod, which will not be deleted automatically. Let's manually delete it so a newer working pod can be started.:

```
$ kubectl --namespace=jhub delete pod jupyter-choldgraf
```

Finally, we'll tell our user to log back in to the JupyterHub. Then let's list our running pods once again:

```
$ kubectl --namespace=jhub get pod
NAME                                READY     STATUS    RESTARTS   AGE
hub-2653507799-r7wf8     1/1       Running   0         3m
jupyter-choldgraf                   1/1       Running   0          18s
proxy-deployment-1227971824-mn2wd   1/1       Running   0          5h
```

And now we see that we have a running user pod!

Note that many debugging situations are not as straightforward as this one. It will take some time before you get a feel for the errors that Kubernetes may throw at you, and how these are tied to your configuration files.

Troubleshooting Examples

The following sections contain some case studies that illustrate some of the more common bugs / gotchas that you may experience using JupyterHub with Kubernetes.

Hub fails to start

Symptom: following `kubectl get pod`, the `hub` pod is in `Error` or `CrashLoopBackoff` state, or appears to be running but accessing the website for the JupyterHub returns an error message in the browser).

Investigating: the output of `kubectl --namespace=jhub logs hub...` shows something like:

```
File "/usr/local/lib/python3.5/dist-packages/jupyterhub/proxy.py", line 589, in get_
↪all_routes
  resp = yield self.api_request('', client=client)
tornado.httpclient.HTTPError: HTTP 403: Forbidden
```

Diagnosis: This is likely because the `hub` pod cannot communicate with the proxy pod API, likely because of a problem in the `secretToken` that was put in `config.yaml`.

Fix: Follow these steps:

1. Create a secret token:

   ```
   openssl rand -hex 32
   ```

2. Add the token to `config.yaml` like so:

   ```
   proxy:
      secretToken: '<output of `openssl rand -hex 32`>'
   ```

3. Redeploy the helm chart:

   ```
   helm upgrade jhub jupyterhub/jupyterhub -f config.yaml
   ```

5.1.3 Authentication

Authentication allows you to control who has access to your JupyterHub deployment. There are many options available to you in controlling authentication, many of which are described below.

Authenticating with OAuth2

JupyterHub's oauthenticator has support for enabling your users to authenticate via a third-party OAuth provider, including GitHub, Google, and CILogon.

Follow the service-specific instructions linked on the oauthenticator repository to generate your JupyterHub instance's OAuth2 client ID and client secret. Then declare the values in the helm chart (`config.yaml`).

Here are example configurations for common authentication services. Note that in each case, you need to get the authentication credential information before you can configure the helm chart for authentication.

GitHub

GitHub is the largest hosting service for git repositories. It is free to create an account at GitHub, and relatively straightforward to set up OAuth credentials so that users can authenticate with their GitHub username/password.

To create OAuth credentials on GitHub, follow these steps:

- Click your profile picture -> settings -> developer settings
- Make sure you're on the "OAuth Apps" tab, then click "New OAuth App"

- Fill out the forms (you'll need your hub address) and generate your ID/Secret.

To enable GitHub authentication, add the following to your `config.yml`:

```
auth:
  type: github
  github:
    clientId: "y0urg1thubc1ient1d"
    clientSecret: "an0ther1ongs3cretstr1ng"
    callbackUrl: "http://<your_jupyterhub_host>/hub/oauth_callback"
```

Make sure that the `callbackUrl` matches the one you set in GitHub.

Giving access to individual GitHub users

The configuration above will allow *any* GitHub user to access your JupyterHub. You can restrict access to a white-list of GitHub users by adding the following to your configuration.

```
auth:
  type: github
  admin:
    access: true
    users:
      - user1
      # ...
  whitelist:
    users:
      - user2
      # ...
```

In this case, `user1` will have *admin* access and `user2` will have regular access to your JupyterHub.

Giving access to organizations on GitHub

You can also restrict access to all of the members of one or more GitHub organizations. To do so, see the configuration below.

```
auth:
  type: github
  github:
    # ...
    orgWhitelist:
      - "SomeOrgName"
  scopes:
    - "read:user"
```

`auth.scopes` can take other values as described in the GitHub Oauth scopes documentation but we recommend `read:user` as this requires no additional configuration by GitHub organisations and users. For example, omitting the scope means members of an organisation must set their membership to Public to login, whereas setting it to `read:org` may require approval of the application by a GitHub organisation admin. Please see this issue for further information.

Note: Changing `auth.scopes` will not change the scope for existing OAuth tokens, you must invalidate them.

Google

Google authentication is used by many universities (it is part of the "G Suite"). Note that using Google authentication requires your Hub to have a domain name (it cannot **only** be accessible via an IP address). For more information on authenticating with Google oauth, see the *Full Example of Google OAuth2*.

```
auth:
  type: google
  google:
    clientId: "yourlongclientidstring.apps.googleusercontent.com"
    clientSecret: "adifferentlongstring"
    callbackUrl: "http://<your_jupyterhub_host>/hub/oauth_callback"
    hostedDomain: "youruniversity.edu"
    loginService: "Your University"
```

CILogon

```
auth:
  type: cilogon
  cilogon:
    clientId: "y0urc1logonc1ient1d"
    clientSecret: "an0ther1ongs3cretstr1ng"
    callbackUrl: "http://<your_jupyterhub_host>/hub/oauth_callback"
```

In order to overcome the caveats of implementing CILogon OAuthAuthenticator for JupyterHub, i.e. default username_claim of ePPN does not work for all providers, e.g. generic OAuth such as Google, Use c.CILogonOAuthenticator.username_claim = 'email' to use email instead of ePPN as the JupyterHub username:

Add to your config.yaml file to inject extra python based configuration that should be in jupyterhub_config.py as below:

```
hub:
  extraConfig:
    myAuthConfig: |
      c.CILogonOAuthenticator.username_claim = 'email'
```

Globus

Globus Auth is a foundational identity and access management platform service designed to address unique needs of the science and engineering community. Globus provides cloud-based services for reliably moving, sharing, publishing and discovering data, whether your files live on a supercomputer, lab cluster, tape archive, public cloud, or your own laptop. Start a Globus app here!

```
auth:
  type: globus
  globus:
    clientId: "y0urc1logonc1ient1d"
    clientSecret: "an0ther1ongs3cretstr1ng"
    callbackUrl: "https://<your_jupyterhub_host>/hub/oauth_callback"
    identityProvider: "youruniversity.edu"
```

Azure Active Directory

Azure Active Directory <https://docs.microsoft.com/en-us/azure/active-directory/>`_ is an identity provider from Microsoft Azure. The main additional option to configure for Azure AD from any other oauth provider is the tenant id.

```
auth:
  type: azuread
  azuread:
    clientId: "your-aad-client-id"
    clientSecret: "your-aad-client-secret"
    tenantId: "your-aad-tenant-id"
    callbackUrl: "https://<your_jupyterhub_host>/hub/oauth_callback"
```

OpenID Connect

OpenID Connect is an identity layer on top of the OAuth 2.0 protocol, implemented by various servers and services. While OpenID Connect endpoint discovery is not supported by oauthentiator, you can still configure JupyterHub to authenticate with OpenID Connect providers by specifying all endpoints in GenericOAuthenticator. By setting `login_service` you can customize the label on the login button.

Here's an example for authenticating against keycloak, after you configure an OIDC Client and obtain the confidential client credentials.

```
hub:
  extraEnv:
    OAUTH2_AUTHORIZE_URL: https://${host}/auth/realms/${realm}/protocol/openid-
↪connect/auth
    OAUTH2_TOKEN_URL: https://${host}/auth/realms/${realm}/protocol/openid-connect/
↪token
    OAUTH_CALLBACK_URL: https://<your_jupyterhub_host>/hub/oauth_callback
auth:
  type: custom
  custom:
    className: oauthenticator.generic.GenericOAuthenticator
    config:
      login_service: "keycloak"
      client_id: "y0urc1logonc1ient1d"
      client_secret: "an0ther1ongs3cretstr1ng"
      token_url: https://${host}/auth/realms/${realm}/protocol/openid-connect/token
      userdata_url: https://${host}/auth/realms/${realm}/protocol/openid-connect/
↪userinfo
      userdata_method: GET
      userdata_params: {'state': 'state'}
      username_key: preferred_username
```

Full Example of Google OAuth2

If your institution is a G Suite customer that integrates with Google services such as Gmail, Calendar, and Drive, you can authenticate users to your JupyterHub using Google for authentication.

Note: Google requires that you specify a fully qualified domain name for your hub rather than an IP address.

1. Log in to the Google API Console.
2. Select a project > Create a project… and set 'Project name'. This is a short term that is only displayed in the console. If you have already created a project you may skip this step.
3. Type "Credentials" in the search field at the top and click to access the Credentials API.
4. Click "Create credentials", then "OAuth client ID". Choose "Application type" > "Web application".
5. Enter a name for your JupyterHub instance. You can give it a descriptive name or set it to be the hub's hostname.
6. Set "Authorized JavaScript origins" to be your hub's URL.
7. Set "Authorized redirect URIs" to be your hub's URL followed by "/hub/oauth_callback". For example, `http://{example.com}/hub/oauth_callback`.
8. When you click "Create", the console will generate and display a Client ID and Client Secret. Save these values.
9. Type "consent screen" in the search field at the top and click to access the OAuth consent screen. Here you will customize what your users see when they login to your JupyterHub instance for the first time. Click Save when you are done.
10. In your helm chart, create a stanza that contains these OAuth fields:

```
auth:
  type: google
  google:
    clientId: "yourlongclientidstring.apps.googleusercontent.com"
    clientSecret: "adifferentlongstring"
    callbackUrl: "http://<your_jupyterhub_host>/hub/oauth_callback"
    hostedDomain: "youruniversity.edu"
    loginService: "Your University"
```

The `callbackUrl` key is set to the authorized redirect URI you specified earlier. Set `hostedDomain` to your institution's domain name. The value of `loginService` is a descriptive term for your institution that reminds your users which account they are using to login.

Authenticating with LDAP

JupyterHub supports LDAP and Active Directory authentication. Read the ldapauthenticator documentation for a full explanation of the available parameters.

Example LDAP Configuration

`auth.ldap.server.address` and `auth.ldap.dn.templates` are required. Other fields are optional.

```
auth:
  type: ldap
  ldap:
    server:
      address: ldap.EXAMPLE.org
    dn:
      templates:
        - 'cn={username},ou=edir,ou=people,ou=EXAMPLE-UNIT,o=EXAMPLE'
```

Example Active Directory Configuration

This example is equivalent to that given in the ldapauthenticator README.

```
auth:
  type: ldap
  ldap:
    server:
      address: ad.EXAMPLE.org
    dn:
      lookup: true
      search:
        filter: '({login_attr}={login})'
        user: 'ldap_search_user_technical_account'
        password: 'secret'
      templates:
        - 'uid={username},ou=people,dc=wikimedia,dc=org'
        - 'uid={username},ou=developers,dc=wikimedia,dc=org'
      user:
        searchBase: 'ou=people,dc=wikimedia,dc=org'
        escape: False
        attribute: 'sAMAccountName'
        dnAttribute: 'cn'
    allowedGroups:
      - 'cn=researcher,ou=groups,dc=wikimedia,dc=org'
      - 'cn=operations,ou=groups,dc=wikimedia,dc=org'
```

Adding a Whitelist

JupyterHub can be configured to only allow a specified whitelist of users to login. This is especially useful if you are using an authenticator with an authentication service open to the general public, such as GitHub or Google.

Note: A whitelist must be used **along with another authenticator**. It simply restricts the usernames that are allowed for your JupyterHub, but is not an authenticator by itself.

You can specify this list of usernames in your `config.yaml`:

```
auth:
  whitelist:
    users:
      - user1
      - user2
```

For example, here's the configuration to use a white list along with the Dummy Authenticator. By default, the Dummy Authenticator will accept any username if they provide the right password. But combining it with a whitelist, users must input **both** an accepted username *and* password.

```
auth:
  type: dummy
  dummy:
    password: 'mypassword'
  whitelist:
    users:
      - user1
      - user2
```

5.1.4 Optimizations

This page contains information and guidelines for improving the reliability, flexibility and stability of your JupyterHub deployment. Many of the settings described is only purposeful for a better autoscaling experience.

To summarize, for a good autoscaling experience, we recommend you to:

- Enable the *continuous image puller*, to prepare added nodes for arriving users.
- Enable *pod priority* and add *user placeholders*, to scale up nodes ahead of real users' arrivals.
- Enable the *user scheduler*, to pack users tight on some nodes and let other nodes become empty and scaled down.
- Set up an autoscaling node pool and dedicate it to user pods by *tainting* the node and requiring user pods, which *tolerate* the nodes' taint, to schedule on these nodes. This way, only user pods can then block scale down.
- Set appropriate user resource *requests* and *limits*, to allow a reasonable amount of users to share a node.

A reasonable final configuration for efficient autoscaling could look something like this:

```
scheduling:
  userScheduler:
    enabled: true
  podPriority:
    enabled: true
  userPlaceholder:
    enabled: true
    replicas: 4
  userPods:
    nodeAffinity:
      matchNodePurpose: require

cull:
  enabled: true
  timeout: 3600
  every: 300

# The resources requested is very important to consider in
# relation to your machine type. If you have a n1-highmem-4 node
# on Google Cloud for example you get 4 cores and 26 GB of
# memory. With the configuration below you would  be able to have
# at most about 50 users per node. This can be reasonable, but it
# may not be, it will depend on your users. Are they mostly
# writing and reading or are they mostly executing code?
singleuser:
  cpu:
    limit: 4
    guarantee: 0.05
  memory:
    limit: 4G
    guarantee: 512M
```

Pulling images before users arrive

If a user pod is scheduled on a node requesting a Docker image that isn't already pulled onto that node, the user will have to wait for it. If the image is large, the wait can be 5 to 10 minutes. This commonly occurs in two situations:

1. A new single-user image is introduced (`helm upgrade`)

With the *hook-image-puller* enabled (the default), the user images being introduced will be pulled to the nodes before the hub pod is updated to utilize the new image. The name hook-image-puller is a technical name referring to how a Helm hook is used to accomplish this, a more informative name would have been *pre-upgrade-image-puller*.

NOTE: With this enabled your `helm upgrade` will take a long time if you introduce a new image as it will wait for the pulling to complete. We recommend that you add `--timeout 600` or similar to your `helm upgrade` command to give it enough time.

The hook-image-puller is enabled by default. To disable it, use the following snippet in your `config.yaml`:

```yaml
prePuller:
  hook:
    enabled: false
```

2. A node is added (Cluster Autoscaler)

   The amount of nodes in a Kubernetes cluster can increase, either by manually scaling up the cluster size or by a cluster autoscaler. As new nodes will come fresh without any images on their disks, a user pod arriving to this node will be forced to wait while the image is pulled.

   With the *continuous-image-puller* enabled (**enabled** by default), the user's container image will be pulled when a new node is added. New nodes can for example be added manually or by a cluster autoscaler. The continuous image-puller uses a daemonset to force Kubernetes to pull the user image on all nodes as soon as a node is present.

   The continuous-image-puller is enabled by default. To disable it, use the following snippet in your `config.yaml`:

```yaml
prePuller:
  continuous:
    # NOTE: if used with a Cluster Autoscaler, also add user-placeholders
    enabled: false
```

   It is important to realize that if the continuous-image-puller together with a Cluster Autoscaler (CA) won't guarantee a reduced wait time for users. It only helps if the CA scales up before real users arrive, but the CA will generally fail to do so. This is because it will only add a node if one or more pods won't fit on the current nodes but would fit more if a node is added, but at that point users are already waiting. To scale up nodes ahead of time we can use *user-placeholders*.

The images that will be pulled

The hook-image-puller and the continuous-image-puller has various sources influencing what images they will pull, as it does in order to prepare nodes ahead of time that may need images. These sources are all found in the values provided with the Helm chart (that can be overridden with `config.yaml`) under the following paths:

Relevant image sources

- `singleuser.image`
- `singleuser.profileList[].kubespawner_override.image`
- `singleuser.extraContainers[].image`
- `prePuller.extraImages.someName`

Additional sources

- `singleuser.networkTools.image`
- `prePuller.pause.image`

For example, with the following configuration, three images would be pulled by the image pullers in order to prepare the nodes that may end up using these images.

```
singleuser:
  image:
    name: jupyter/minimal-notebook
    tag: 2343e33dec46
  profileList:
    - display_name: "Minimal environment"
      description: "To avoid too much bells and whistles: Python."
      default: true
    - display_name: "Datascience environment"
      description: "If you want the additional bells and whistles: Python, R, and␣
↪Julia."
      kubespawner_override:
        image: jupyter/datascience-notebook:2343e33dec46

prePuller:
  extraImages:
    myOtherImageIWantPulled:
      name: jupyter/all-spark-notebook
      tag: 2343e33dec46
```

Efficient Cluster Autoscaling

A *Cluster Autoscaler* (CA) will help you add and remove nodes from the cluster. But the CA needs some help to function well. Without help, it will both fail to scale up before users arrive and scale down nodes aggressively enough without disrupting users.

Scaling up in time (user placeholders)

A *Cluster Autoscaler* (CA) will add nodes when pods don't fit on available nodes but would fit if another node is added. But, this may lead to a long waiting time for the pod, and as a pod can represent a user, it can lead to a long waiting time for a user. There are now options to address this.

With Kubernetes 1.11+ (that requires Helm 2.11+), Pod Priority and Preemption was introduced. This allows pods with higher priority to preempt / evict pods with lower priority if that would help the higher priority pod fit on a node.

This priority mechanism allows us to add dummy users or *user-placeholders* with low priority that can take up resources until a real user with (higher priority) requires it. At this time, the lower priority pod will get preempted to make room for the high priority pod. This now evicted user-placeholder will now be able to signal to the CA that it needs to scale up.

The user placeholders will have the same resources requests as the default user. This means that if you have three user placeholders running, real users will only need to wait for a scale up if more than three users arrive in an interval of time less than it takes to make a node ready for use.

To use three user placeholders for example, that can do their job thanks to pod priority, add the following configuration:

```
scheduling:
  podPriority:
    enabled: true
  userPlaceholder:
    # Specify three dummy user pods will be used as placeholders
    replicas: 3
```

For further discussion about user placeholders, see @MinRK's excellent post where he analyzed its introduction on mybinder.org.

IMPORTANT: Further settings may be required for successful use of the pod priority depending on how your cluster autoscaler is configured. This is known to work on GKE, but we don't know how it works on other cloud providers or kubernetes. See the configuration reference for more details.

Scaling down efficiently

Scaling up is the easy part, scaling down is harder. To scale down a node, certain technical criteria need to be met. The central one is in order for a node to be scaled down, it must be free from pods that aren't allowed to be disrupted. Pods that are not allowed to be disrupted are, for example, real user pods, important system pods, and some JupyterHub pods (without a permissive PodDisruptionBudget). Consider for example that many users arrive to your JupyterHub during the daytime. New nodes are added by the CA. Some system pod ends up on the new nodes along with the user pods for some reason. At night when the *culler* has removed many inactive pods from some nodes. They are now free from user pods but there is still a single system pod stopping the CA from removing the node.

To avoid these scale down failures, we recommend using a *dedicated node pool* for the user pods. That way, all the important system pods will run at one or a limited set of nodes, so the autoscaling user nodes can scale from 0 to X and back from X to 0.

This section about scaling down efficiently, will also explains how the *user scheduler* can help you reduce the failures to scale down due to blocking user pods.

Using a dedicated node pool for users

To set up a dedicated node pool for user pods, we can use *taints and tolerations*. If we add a taint to all the nodes in the node pool, and a toleration on the user pods to tolerate being scheduled on a tainted node, we have practically dedicated the node pool to be used only by user pods.

To make user pods schedule on a dedicated node for them, you need to do the following:

1. Setup a node pool (with autoscaling), a certain label, and a certain taint.

   If you need help on how to do this, please refer to your cloud providers documentation. A node pool may be called a node group.

   - The label: `hub.jupyter.org/node-purpose=user`

     NOTE: Cloud providers often have their own labels, separate from kubernetes labels, but this label must be a kubernetes label.

   - The taint: `hub.jupyter.org/dedicated=user:NoSchedule`

     NOTE: You may need to replace `/` with `_` due cloud provider limitations. Both taints are tolerated by the user pods.

2. Make user pods require to be scheduled on the node pool setup above

   If you don't require the user pods to schedule on their dedicated node, you may fill up the nodes where the other software runs. This can cause a `helm upgrade` command to fail. For example, you may have run out of resources for non-user pods that cannot schedule on the autoscaling node pool as they need during a rolling update.

The default setting is to make user pods *prefer* to be scheduled on nodes with the `hub.jupyter.org/node-purpose=user` label, but you can also make it *required* using the configuration below.

```yaml
scheduling:
  userPods:
    nodeAffinity:
      # matchNodePurpose valid options:
      # - ignore
      # - prefer (the default)
      # - require
      matchNodePurpose: require
```

NOTE: If you end up *not* using a dedicated node pool for users and want to scale down efficiently, you will need to learn about PodDisruptionBudget resources and do quite a bit more work in order to avoid ending up with almost empty nodes not scaling down.

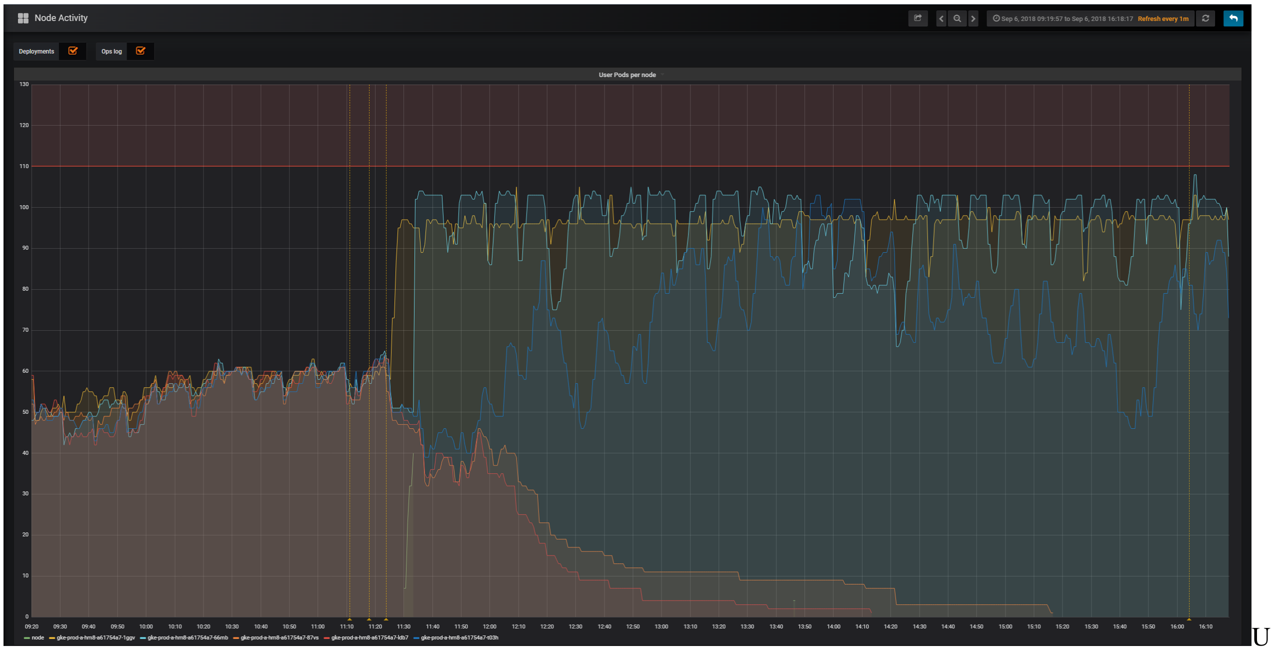
Using available nodes efficiently (the user scheduler)

If you have users starting new servers while the total number of active users decreasing, how will you free up a node so it can be scaled down?

This is what the *user scheduler* helps you with. The user scheduler's only task is to schedule new user pods to the *most utilized node*. This can be compared to the *default scheduler* that instead always tries to schedule pods so the *least utilized node*. Only the user scheduler would allow the underutilized nodes to free up over time as the total amount of users decrease but a few users still arrive.

NOTE: If you don't want to scale down the nodes you have, it would make more sense to let the users spread out and utilize all available nodes. Only activate the user scheduler if you have an autoscaling node pool.

To see the user scheduler in action, look at the following graph from the mybinder.org deployment. The graph is from when the user scheduler was enabled for the first time, it is showing the amount of user pods active on five different nodes. When the user scheduler was enabled, two nodes were in time freed up from user pods and scaled down.



User

scheduler node activity

To enable the user scheduler:

```yaml
scheduling:
  userScheduler:
    enabled: true
```

NOTE: For the user scheduler to work well, you need old user pods to shut down at some point. Make sure to properly configure the *culler*.

5.1.5 Security

The information in this document focuses primarily on cloud based deployments. For on-premise deployments, additional security work that is specific to your installation method would also be required. Note that your specific installation's security needs might be more or less stringent than what we can offer you here.

Brad Geesamen gave a wonderful talk titled Hacking and Hardening Kubernetes by Example at Kubecon NA 2017. You can watch the talk or read the slides. Highly recommended that you do so to understand the security issues you are up against when using Kubernetes to run JupyterHub.

Reporting a security issue

If you find a security vulnerability in JupyterHub, either a failure of the code to properly implement the model described here, or a failure of the model itself, please report it to security@ipython.org.

If you prefer to encrypt your security reports, you can use this PGP public key.

HTTPS

This section describes how to enable HTTPS on your JupyterHub. The easiest way to do so is by using Let's Encrypt, though we'll also cover how to set up your own HTTPS credentials. For more information on HTTPS security see the certificates section of this blog post.

Set up your domain

1. Buy a domain name from a registrar. Pick whichever one you want.
2. Create an A record from the domain you want to use, pointing to the EXTERNAL-IP of the proxy-public service. The exact way to do this will depend on the DNS provider that you're using.
3. Wait for the change to propagate. Propagation can take several minutes to several hours. Wait until you can type in the name of the domain you bought and it shows you the JupyterHub landing page.

   It is important that you wait - prematurely going to the next step might cause problems!

Set up automatic HTTPS

JupyterHub uses Let's Encrypt to automatically create HTTPS certificates for your deployment. This will cause your HTTPS certificate to automatically renew every few months. To enable this, make the following changes to your `config.yaml` file:

1. Specify the two bits of information that we need to automatically provision HTTPS certificates - your domain name & a contact email address.

```
proxy:
  https:
    hosts:
      - <your-domain-name>
    letsencrypt:
      contactEmail: <your-email-address>
```

2. Apply the config changes by running `helm upgrade ...`
3. Wait for about a minute, now your hub should be HTTPS enabled!

Set up manual HTTPS

If you have your own HTTPS certificates & want to use those instead of the automatically provisioned Let's Encrypt ones, that's also possible. Note that this is considered an advanced option, so we recommend not doing it unless you have good reasons.

There are two ways to specify your manual certificate, directly in the config.yaml or by creating a Kubernetes `secret`.

Specify certificate in config.yaml

1. Add your domain name & HTTPS certificate info to your `config.yaml`

```
proxy:
  https:
    hosts:
      - <your-domain-name>
    type: manual
    manual:
      key: |
        -----BEGIN RSA PRIVATE KEY-----
        ...
        -----END RSA PRIVATE KEY-----
      cert: |
        -----BEGIN CERTIFICATE-----
        ...
        -----END CERTIFICATE-----
```

2. Apply the config changes by running helm upgrade ....
3. Wait for about a minute, now your hub should be HTTPS enabled!

Specify certificate through Secret resource

1. Create a `secret` resource with type `kubernetes.io/tls` containing your certificate.

   `kubectl create secret tls example-tls --key="tls.key" --cert="tls.crt"`

2. Add your domain and the name of your `secret` to your config.yaml.

```
proxy:
  https:
    hosts:
      - <your-domain-name>
    type: secret
```

(continues on next page)

(continued from previous page)

```
      secret:
        name: example-tls
```

3. Apply the config changes by running helm upgrade ....
4. Wait for about a minute, now your hub should be HTTPS enabled!

Off-loading SSL to a Load Balancer

In some environments with a trusted network, you may want to terminate SSL at a load balancer. If https is enabled, and `proxy.https.type` is set to `offload`, the HTTP and HTTPS front ends target the HTTP port from JupyterHub.

The HTTPS listener on the load balancer will need to be configured based on the provider. If you're using AWS and a certificate provided by their certificate manager, your config.yml might look something like:

```
proxy:
  https:
    enabled: true
    type: offload
  service:
    annotations:
      # Certificate ARN
      service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "arn:aws:acm:us-east-
↪1:1234567891011:certificate/uuid"
      # The protocol to use on the backend, we use TCP since we're using websockets
      service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "tcp"
      # Which ports should use SSL
      service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "https"
      service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout: "3600"
```

Annotation options will vary by provider. Kubernetes provides a list for popular cloud providers in their documentation.

Confirm that your domain is running HTTPS

There are many ways to confirm that a domain is running trusted HTTPS certificates. One options is to use the Qualys SSL Labs security report generator. Use the following URL structure to test your domain:

```
http://ssllabs.com/ssltest/analyze.html?d=<YOUR-DOMAIN>
```

Secure access to Helm

In its default configuration, helm pretty much allows root access to all other pods running in your cluster. See this Bitnami Helm security article for more information. As a consequence, the default allows all users in your cluster to pretty much have root access to your whole cluster!

You can mitigate this by limiting public access to the Tiller API. To do so, use the following command:

```
kubectl --namespace=kube-system patch deployment tiller-deploy --type=json --patch='[{
↪"op": "add", "path": "/spec/template/spec/containers/0/command", "value": ["/tiller
↪", "--listen=localhost:44134"]}]'
```

This limit shouldn't affect helm functionality in any form.

Audit Cloud Metadata server access

Most cloud providers have a static IP you can hit from any of the compute nodes, including the user pod, to get metadata about the cloud. This metadata can contain very sensitive info, and this metadata, in the wrong hands, can allow attackers to take full control of your cluster and cloud resources. It is **critical** to secure the metadata service. We block access to this IP by default (as of v0.6), so you are protected from this!

The slides beginning at *Slide 38* provides more information on the dangers presented by this attack.

If you need to enable access to the metadata server for some reason, you can do the following in config.yaml:

```yaml
singleuser:
  cloudMetadata:
    enabled: true
```

Delete the Kubernetes Dashboard

The Kubernetes Dashboard gets created by default in many installations. Although the Dashboard contains useful information, the Dashboard also poses a security risk. We **recommend** deleting it and not using it for the time being until the Dashboard becomes properly securable.

You can mitigate this by deleting the Kubernetes Dashboard deployment from your cluster. This can be most likely performed with:

```
kubectl --namespace=kube-system delete deployment kubernetes-dashboard
```

In older clusters, you might have to do:

```
kubectl --namespace=kube-system delete rc kubernetes-dashboard
```

Use Role Based Access Control (RBAC)

Kubernetes supports, and often requires, using Role Based Access Control (RBAC) to secure which pods / users can perform what kinds of actions on the cluster. RBAC rules can be set to provide users with minimal necessary access based on their administrative needs.

It is **critical** to understand that if RBAC is disabled, all pods are given `root` equivalent permission on the Kubernetes cluster and all the nodes in it. This opens up very bad vulnerabilites for your security.

As of the Helm chart v0.5 used with JupyterHub and BinderHub, the helm chart can natively work with RBAC enabled clusters. To provide sensible security defaults, we ship appropriate minimal RBAC rules for the various components we use. We **highly recommend** using these minimal or more restrictive RBAC rules.

If you want to disable the RBAC rules, for whatever reason, you can do so with the following snippet in your `config.yaml`:

```yaml
rbac:
  enabled: false
```

We strongly **discourage disabling** the RBAC rules and remind you that this action will open up security vulnerabilities. However, some cloud providers may not support RBAC in which case you can disable it with this config.

Kubernetes API Access

Allowing direct user access to the Kubernetes API can be dangerous. It allows users to grant themselves more privileges, access other users' content without permission, run (unprofitable) bitcoin mining operations & various other not-legitimate activities. By default, we do not allow access to the service account credentials needed to access the Kubernetes API from user servers for this reason.

If you want to (carefully!) give access to the Kubernetes API to your users, you can do so with the following in your `config.yaml`:

```
singleuser:
  serviceAccountName: <service-account-name>
```

You can either manually create a service account for use by your users and specify the name of that here (recommended) or use `default` to give them access to the default service account for the namespace. You should ideally also (manually) set up RBAC rules for this service account to specify what permissions users will have.

This is a sensitive security issue (similar to writing sudo rules in a traditional computing environment), so be very careful.

There's ongoing work on making this easier!

Kubernetes Network Policies

Kubernetes has optional support for network policies which lets you restrict how pods can communicate with each other and the outside world. This can provide additional security within JupyterHub, and can also be used to limit network access for users of JupyterHub.

By default, the JupyterHub helm chart **disables** network policies.

Enabling network policies

Important: If you decide to enable network policies, you should be aware that a Kubernetes cluster may have partial, full, or no support for network policies. Kubernetes will **silently ignore** policies that aren't supported. Please use **caution** if enabling network policies and verify the policies behave as expected, especially if you rely on them to restrict what users can access.

You can enable network policies in your `config.yaml`:

```
hub:
  networkPolicy:
    enabled: true
proxy:
  networkPolicy:
    enabled: true
singleuser:
  networkPolicy:
    enabled: true
```

The default singleuser policy allows all outbound network traffic, meaning JupyterHub users are able to connect to all resources inside and outside your network. To restrict outbound traffic to DNS, HTTP and HTTPS:

```
singleuser:
  networkPolicy:
    enabled: true
    egress:
```

(continues on next page)

(continued from previous page)

```
      - ports:
          - port: 53
            protocol: UDP
      - ports:
          - port: 80
            protocol: TCP
      - ports:
          - port: 443
            protocol: TCP
```

See the Kubernetes documentation for further information on defining policies.

Restricting Load Balancer Access

By default any IP address can access your JupyterHub deployment through the load balancer service. In case you want to restrict which IP addresses are allowed to access the load balancer, you can specify a list of IP CIDR addresses in your `config.yaml` as follows:

```
proxy:
  service:
    loadBalancerSourceRanges:
      - 111.222.333.444/32
      - 222.333.444.555/32
```

This would restrict the access to only two IP addresses: `111.222.333.444` and `222.333.444.555`.

For more information please check the Kubernetes documentation on restricting load balancer access.

5.1.6 Upgrading your Helm chart

This page covers best-practices in upgrading your JupyterHub deployment via updates to the Helm Chart.

Upgrading from one version of the Helm Chart to the next should be as seamless as possible, and generally shouldn't require major changes to your deployment. Check the CHANGELOG for each release to find out if there are any breaking changes in the newest version.

For additional help, feel free to reach out to us on gitter or the mailing list!

Major helm-chart upgrades

These steps are **critical** before performing a major upgrade.

1. Always backup your database!
2. Review the CHANGELOG for incompatible changes and upgrade instructions.
3. Update your configuration accordingly.
4. User servers may need be stopped prior to the upgrade, or restarted after it.
5. If you are planning an upgrade of a critical major installation, we recommend you test the upgrade out on a staging cluster first before applying it to production.

v0.5 to v0.6

See the CHANGELOG.

v0.4 to v0.5

Release 0.5 contains a major JupyterHub version bump (from 0.7.2 to 0.8). Since it is a major upgrade of JupyterHub that changes how authentication is implemented, user servers must be stopped during the upgrade. The database schema has also changed, so a database upgrade must be performed.

See the documentation for v0.5 for the upgrade process as well as the CHANGELOG for this release for more information about changes.

Subtopics

This section covers upgrade information specific to the following:

- `helm upgrade` command
- Databases
- RBAC (Role Based Access Control)
- Custom Docker images

`helm upgrade` command

After modifying your `config.yaml` file according to the CHANGELOG, you will need `<YOUR-HELM-RELEASE-NAME>` to run the upgrade commands. To find `<YOUR-RELEASE-NAME>`, run:

```
helm list
```

Make sure to test the upgrade on a staging environment before doing the upgrade on a production system!

To run the upgrade:

```
helm upgrade <YOUR-HELM-RELEASE-NAME> jupyterhub/jupyterhub --version=<RELEASE-
↪VERSION> -f config.yaml
```

For example, to upgrade to v0.6, enter and substituting `<YOUR-HELM-RELEASE-NAME>` and version v0.6:

```
helm upgrade <YOUR-HELM-RELEASE-NAME> jupyterhub/jupyterhub --version=v0.6 -f config.
↪yaml
```

Database

This release contains a major JupyterHub version bump (from 0.7.2 to 0.8). If you are using the default database provider (SQLite), then the required db upgrades will be performed automatically when you do a `helm upgrade`.

Default (SQLite): The database upgrade will be performed automatically when you *perform the upgrade*

MySQL / PostgreSQL: You will execute the following steps, which includes a manual update of your database:

1. Make a full backup of your database, just in case things go bad.

2. Make sure that the database user used by JupyterHub to connect to your database can perform schema migrations like adding new tables, altering tables, etc.

3. In your `config.yaml`, add the following config:

   ```
   hub:
     db:
       upgrade: true
   ```

4. Do a *helm upgrade*. This should perform the database upgrade needed.

5. Remove the lines added in step 3, and do another *helm upgrade*.

Role based access control

RBAC is the user security model in Kubernetes that gives applications only as much access they need to the kubernetes API and not more. Prior to this, applications were all running with the equivalent of root on your Kubernetes cluster. This release adds appropriate roles for the various components of JupyterHub, for much better ability to secure clusters.

RBAC is turned on by default. But, if your cluster is older than 1.8, or you have RBAC enforcement turned off, you might want to explicitly disable it. You can do so by adding the following snippet to your `config.yaml`:

```
rbac:
  enabled: false
```

This is especially true if you get an error like:

```
Error: the server rejected our request for an unknown reason (get clusterrolebindings.
 ↪rbac.authorization.k8s.io)
```

when doing the upgrade!

Custom Docker Images: JupyterHub version match

If you are using a custom built image, make sure that the version of the JupyterHub package installed in it is now 0.8.1. It needs to be 0.8.1 for it to work with v0.6 of the helm chart.

For example, if you are using `pip` to install JupyterHub in your custom Docker Image, you would use:

```
RUN pip install --no-cache-dir jupyterhub==0.8.1
```

JupyterHub versions installed in each Helm Chart

Each Helm Chart is packaged with a specific version of JupyterHub (and other software as well). See the Helm Chart repository for information about the versions of relevant software packages.

Troubleshooting

If the upgrade is failing on a test system or a system that does not serve users, you can try deleting the helm chart using:

```
helm delete <YOUR-HELM-RELEASE-NAME> --purge
```

`helm list` may be used to find .

5.1.7 FAQ

This section contains frequently asked questions about the JupyterHub deployment. For information on debugging Kubernetes, see *Debugging*.

I thought I had deleted my cloud resources, but they still show up. Why?

You probably deleted the specific nodes, but not the Kubernetes cluster that was controlling those nodes. Kubernetes is designed to make sure that a specific set of resources is available at all times. This means that if you only delete the nodes, but not the Kubernetes instance, then it will detect the loss of computers and will create two new nodes to compensate.

How does billing for this work?

JupyterHub isn't handling any of the billing for your usage. That's done through whatever cloud service you're using. For considerations about managing cost with JupyterHub, see *Appendix: Projecting deployment costs*.

What version of JupyterHub is installed in the Helm Chart?

Each Helm Chart is packaged with a specific version of JupyterHub (and other software as well). See see the Helm Chart repository for information about the versions of relevant software packages.

5.1.8 Advanced Topics

This page contains a grab bag of various useful topics that don't have an easy home elsewhere:

- Ingress
- Arbitrary extra code and configuration in `jupyterhub_config.py`

Most people setting up JupyterHubs on popular public clouds should not have to use any of this information, but these topics are essential for more complex installations.

Ingress

If you are using a Kubernetes Cluster that does not provide public IPs for services directly, you need to use an ingress to get traffic into your JupyterHub. This varies wildly based on how your cluster was set up, which is why this is in the 'Advanced' section.

You can enable the required `ingress` object with the following in your `config.yaml`

```
ingress:
  enabled: true
  hosts:
    - <hostname>
```

You can specify multiple hosts that should be routed to the hub by listing them under `ingress.hosts`.

Note that you need to install and configure an Ingress Controller for the ingress object to work.

We recommend the community-maintained nginx-ingress controller, **kubernetes/ingress-nginx**. Note that Nginx maintains two additional ingress controllers. For most use cases, we recommend the community maintained **kubernetes/ingress-nginx** since that is the ingress controller that the development team has the most experience using.

Ingress and Automatic HTTPS with kube-lego & Let's Encrypt

When using an ingress object, the default automatic HTTPS support does not work. To have automatic fetch and renewal of HTTPS certificates, you must set it up yourself.

Here's a method that uses kube-lego to automatically fetch and renew HTTPS certificates from Let's Encrypt. This approach with kube-lego and Let's Encrypt currently only works with two ingress controllers: the community-maintained **kubernetes/ingress-nginx** and **google cloud's ingress controller**.

1. Make sure that DNS is properly set up (configuration depends on the ingress controller you are using and how your cluster was set up). Accessing `<hostname>` from a browser should route traffic to the hub.
2. Install & configure kube-lego using the kube-lego helm-chart. Remember to change `config.LEGO_EMAIL` and `config.LEGO_URL` at the least.
3. Add an annotation + TLS config to the ingress so kube-lego knows to get certificates for it:

```
ingress:
  annotations:
    kubernetes.io/tls-acme: "true"
  tls:
   - hosts:
      - <hostname>
     secretName: kubelego-tls-jupyterhub
```

This should provision a certificate, and keep renewing it whenever it gets close to expiry!

Arbitrary extra code and configuration in `jupyterhub_config.py`

Sometimes the various options exposed via the helm-chart's `values.yaml` is not enough, and you need to insert arbitrary extra code / config into `jupyterhub_config.py`. This is a valuable escape hatch for both prototyping new features that are not yet present in the helm-chart, and also for installation-specific customization that is not suited for upstreaming.

There are four properties you can set in your `config.yaml` to do this.

`hub.extraConfig`

The value specified for `hub.extraConfig` is evaluated as python code at the end of `jupyterhub_config.py`. You can do anything here since it is arbitrary Python Code. Some examples of things you can do:

1. Override various methods in the Spawner / Authenticator by subclassing them. For example, you can use this to pass authentication credentials for the user (such as GitHub OAuth tokens) to the environment. See the JupyterHub docs for an example.
2. Specify traitlets that take callables as values, allowing dynamic per-user configuration.
3. Set traitlets for JupyterHub / Spawner / Authenticator that are not currently supported in the helm chart

Unfortunately, you have to write your python *in* your YAML file. There's no way to include a file in `config.yaml`.

You can specify `hub.extraConfig` as a raw string (remember to use the `|` for multi-line YAML strings):

```
hub:
  extraConfig: |
    import time
    c.Spawner.environment += {
```

(continues on next page)

(continued from previous page)

```
        "CURRENT_TIME": str(time.time())
    }
```

You can also specify `hub.extraConfig` as a dictionary, if you want to logically split your customizations. The code will be evaluated in alphabetical sorted order of the key.

```
hub:
  extraConfig:
    00-first-config: |
      # some code
    10-second-config: |
      # some other code
```

`custom` configuration

The contents of `values.yaml` is passed through to the Hub image. You can access these values via the `z2jh.get_config` function, for further customization of the hub pod. Version 0.8 of the chart adds a top-level `custom` field for passing through additional configuration that you may use. It can be arbitrary YAML. You can use this to separate your code (which goes in `hub.extraConfig`) from your config (which should go in `custom`).

For example, if you use the following snippet in your config.yaml file:

```
custom:
  myString: Hello!
  myList:
    - Item1
    - Item2
  myDict:
    key: value
  myLongString: |
    Line1
    Line2
```

In your `hub.extraConfig`,

1. `z2jh.get_config('custom.myString')` will return a string `"Hello!"`
2. `z2jh.get_config('custom.myList')` will return a list `["Item1", "Item2"]`
3. `z2jh.get_config('custom.myDict')` will return a dict `{"key": "value"}`
4. `z2jh.get_config('custom.myLongString')` will return a string `"Line1\nLine2"`
5. `z2jh.get_config('custom.nonExistent')` will return `None` (since you didn't specify any value for `nonExistent`)
6. `z2jh.get_config('custom.myDefault', True)` will return `True`, since that is specified as the second parameter (default)

You need to have a `import z2jh` at the top of your `extraConfig` for `z2jh.get_config()` to work.

Changed in version 0.8: `hub.extraConfigMap` used to be required for specifying additional values to pass, which was more restrictive. `hub.extraConfigMap` is deprecated in favor of the new top-level `custom` field, which allows fully arbitrary yaml.

`hub.extraEnv`

This property takes a dictionary that is set as environment variables in the hub container. You can use this to either pass in additional config to code in your `hub.extraConfig` or set some hub parameters that are not settable by other means.

`hub.extraContainers`

A list of extra containers that are bundled alongside the hub container in the same pod. This is a common pattern in kubernetes that as a long list of cool use cases. Some example use cases are:

1. Database Proxies, which are sometimes required for the hub to talk to its configured database (in Google Cloud) for example
2. Servers / other daemons that are used by code in your `hub.customConfig`

The items in this list must be valid kubernetes container specifications.

Specifying suitable hub storage

By default, the hub's sqlite-pvc setting will dynamically create a disk to store the sqlite database. It is possible to configure other storage classes under hub.db.pvc, but make sure to choose one that the hub can write quickly and safely to. Slow or higher latency storage classes can cause hub operations to lag which may ultimately lead to HTTP errors in user environments.

5.1.9 Appendix: Projecting deployment costs

Important: Clarification on cost projections

As a non-profit research project, Project Jupyter does not offer, recommend, or sell cloud deployment services for JupyterHub.

The information in this section is offered as guidance as requested by our users. We **caution** that costs can vary widely based on providers selected and your use cases.

Cost calculators for cloud providers

Below are several links to cost estimators for cloud providers:

- Google Cloud Platform cost calculator
- Amazon AWS cost calculator
- Microsoft Azure cost claculator

Factors influencing costs

Cost estimates depend highly on your deployment setup. Several factors that significantly influence cost estimates, include:

- Computational resources provided to users

- Number of users
- Usage patterns of users

Computational Resources

Memory (RAM) makes up the largest part of a cost estimate. More RAM means that your users will be able to work with larger datasets with more flexibility, but it can also be expensive.

Persistent storage for users, if needed, is another element that will impact the cost estimate. If users don't have persistent storage, then disks will be wiped after users finish their sessions. None of their changes will be saved. This requires significantly fewer storage resources, and also results in faster load times.

For an indicator of how costs scale with computational resources, see the Google Cloud pricing page.

Users

The number of users has a direct relationship to cost estimates. Since a deployment may support different types of users (i.e. researchers, students, instructors) with varying hardware and storage needs, take into account both the type of users and the number per type.

User usage patterns

Another important factor is what usage pattern your users will have. Will they all use the JupyterHub at once, such as during a large class workshop? will users use JupyterHub at different times of day?

The usage patterns and peak load on the system have important implications for the resources you need to provide. In the future JupyterHub will have auto-scaling functionality, but currently it does not. This means that you need to provision resources for the *maximum* expected number of users at one time.

Interactive Cost Estimator (rough estimate)

This small notebook may help you to make an initial planning estimate of costs for your JupyterHub instance.

To use the estimator, the button below will take you to an interactive notebook served with Binder. Run the cells and you'll be able to choose the computational requirements you have, as well as draw a pattern of usage you expect over time. It will estimate the costs for you.

Warning: **The cost estimator is a very rough estimate.** It is based on Google Cloud Engine instances served from Oregon. Costs will vary based on your location / provider, and will be highly variable if you implement any kind of auto-scaling. Treat it as an order-of-magnitude estimate, not a hard rule.

Examples

Here are a few examples that describe different use cases and the amount of resources used by a particular JupyterHub implementation. There are many factors that go into these estimates, and you should expect that your actual costs may vary significantly under other conditions.

Data 8

The Data 8 course at UC Berkeley used a JupyterHub to coordinate all course material and to provide a platform where students would run their code. This consisted of many hundreds of students, who had minimal requirements in terms of CPU and memory usage. Ryan Lovett put together a short Jupyter notebook estimating the cost for computational resources depending on the student needs.

CHAPTER 6

Resources from the community

This section gives the community a space to provide information on setting up, managing, and maintaining JupyterHub.

Important: We recognize that Kubernetes has many deployment options. As a project team with limited resources to provide end user support, we rely on community members to share their collective Kubernetes knowledge and JupyterHub experiences.

Note: **Contributing to Z2JH**. If you would like to help improve the Zero to JupyterHub guide, please see the issues page as well as the contributor guide.

We hope that you will use this section to share deployments with on a variety of infrastructure and for different use cases. There is also a *community maintained list* of users of this Guide and the JupyterHub Helm Chart.

Please submit a pull request to add to this section. Thanks.

6.1 Community Resources

This section gives the community a space to provide information on setting up, managing, and maintaining JupyterHub.

Important: We recognize that Kubernetes has many deployment options. As a project team with limited resources to provide end user support, we rely on community members to share their collective Kubernetes knowledge and JupyterHub experiences.

Note: **Contributing to Z2JH**. If you would like to help improve the Zero to JupyterHub guide, please see the issues page as well as the contributor guide.

We hope that you will use this section to share deployments with on a variety of infrastructure and for different use cases. There is also a *community maintained list* of users of this Guide and the JupyterHub Helm Chart.

Please submit a pull request to add to this section. Thanks.

6.1.1 Tips and Snippets

This is a page to collect a few particularly useful patterns and snippets that help you interact with your Kubernetes cluster and JupyterHub. If there's something that you think is generic enough (and not obvious enough) to be added to this page, please feel free to make a PR!

`kubectl` autocompletion

Kubernetes has a helper script that allows you to auto-complete commands and references to objects when using `kubectl`. This lets you `TAB`-complete and saves a lot of time.

Here are the instructions to install kubectl auto-completion.

`helm` autocompletion

Helm also has an auto-completion script that lets you `TAB`-complete your commands when using Helm.

Here are the instructions to install helm auto-completion.

Managing `kubectl` contexts

Oftentimes people manage multiple Kubernetes deployments at the same time. `kubectl` handles this with the idea of "contexts", which specify which Kubernetes deployment you are referring to when you type `kubectl get XXX`.

To see a list of contexts currently available to you, use the following command:

```
kubectl config get-contexts
```

This will list all of your Kubernetes contexts. You can select a particular context by entering:

```
kubectl config use-context <CONTEXT-NAME>
```

Specifying a default namespace for a context

If you grow tired of typing `namespace=XXX` each time you type a kubernetes command, here's a snippet that will allow you set a default namespace for a given Kubernetes context:

```
kubectl config set-context $(kubectl config current-context) \
        --namespace=<YOUR-NAMESPACE>
```

The above command will only apply to the currently active context, and will allow you to skip the `--namespace=` part of your commands for this context.

Using labels and selectors with `kubectl`

Sometimes it's useful to select an entire class of Kubernetes objects rather than referring to them by their name. You can attach an arbitrary set of labels to a Kubernetes object, and can then refer to those labels when searching with `kubectl`.

To search based on a label value, use the `-l` or `--selector=` keyword arguments. For example, JupyterHub creates a specific subset of labels for all user pods. You can search for all user pods with the following label query:

```
kubectl --namespace=<YOUR-NAMESPACE> get pod \
        -l "component=singleuser-server"
```

For more information, see the Kubernetes labels and selectors page.

Asking for a more verbose or structured output

Sometimes the information that's in the default output for `kubectl get <XXX>` is not enough for your needs, or isn't structured the way you'd like. We recommend looking into the different Kubernetes output options, which can be modified like so:

```
kubectl --namespace=<NAMESPACE> get pod -o <json|yaml|wide|name...>
```

You can find more information on what kinds of output you can generate at the kubectl information page. (click and search for the text "Output Options")

6.1.2 Used by

This is a community maintained list of organizations / people using the Zero to JupyterHub guide and Helm chart to maintain their JupyterHub. Send us a Pull Request to add yourself to this alphabetically sorted list!

- Data Science Education Program's DataHub at University of California, Berkeley
- MyBinder.org
- PAWS at Wikimedia Cloud Services
- A group of universities near Paris (led by a team at CNRS and Polytechnique) deployed a cross-institutional JupyterHub for university use.
- Globus runs an instance of Zero-to-JupyterHub to help users learn about Globus REST APIs.

6.1.3 Additional Resources

This page contains links and references to other material in the JupyterHub ecosystem. It may include other guides, examples of JupyterHub deployments, or posts from the community.

If you have a suggestion for something to add to this page, please open an issue.

Links to community project resources

- **Automatically deploy a JupyterHub on AWS** from the Space Telescope Software Institute * github repository * blog post

CHAPTER 7

Reference

7.1 Reference

7.1.1 Configuration Reference

The JupyterHub Helm chart is configurable by values in your config.yaml. In this way, you can extend user resources, build off of different Docker images, manage security and authentication, and more.

Below is a description of many *but not all* of the configurable values for the Helm chart. To see *all* configurable options, inspect their default values defined here.

For more guided information about some specific things you can do with modifications to the helm chart, see the *Customization Guide*.

hub

hub.cookieSecret

A 32-byte cryptographically secure randomly generated string used to sign values of secure cookies set by the hub. If unset, jupyterhub will generate one on startup and save it in the file jupyterhub_cookie_secret in the /srv/jupyterhub directory of the hub container. A value set here will make JupyterHub overwrite any previous file.

You do not need to set this at all if you are using the default configuration for storing databases - sqlite on a persistent volume (with hub.db.type set to the default sqlite-pvc). If you are using an external database, then you must set this value explicitly - or your users will keep getting logged out each time the hub pod restarts.

Changing this value will all user logins to be invalidated. If this secret leaks, *immediately* change it to something else, or user data can be compromised

```
# to generate a value, run
openssl rand -hex 32
```

hub.imagePullPolicy

Set the imagePullPolicy on the hub pod.

See the Kubernetes docs for more info on what the values mean.

hub.imagePullSecret

Creates an image pull secret for you and makes the hub pod utilize it, allowing it to pull images from private image registries.

Using this configuration option automates the following steps that normally is required to pull from private image registries.

```
# you won't need to run this manually...
kubectl create secret docker-registry hub-image-credentials \
  --docker-server=<REGISTRY> \
  --docker-username=<USERNAME> \
  --docker-email=<EMAIL> \
  --docker-password=<PASSWORD>
```

```
# you won't need to specify this manually...
spec:
  imagePullSecrets:
    - name: hub-image-credentials
```

To learn the username and password fields to access a gcr.io registry from a Kubernetes cluster not associated with the same google cloud credentials, look into this guide and read the notes about the password.

hub.imagePullSecret.enabled

Enable the creation of a Kubernetes Secret containing credentials to access a image registry. By enabling this, the hub pod will also be configured to use these credentials when it pulls its container image.

hub.imagePullSecret.registry

Name of the private registry you want to create a credential set for. It will default to Docker Hub's image registry.

Examples:

- https://index.docker.io/v1/
- quay.io
- eu.gcr.io
- alexmorreale.privatereg.net

hub.imagePullSecret.username

Name of the user you want to use to connect to your private registry. For external gcr.io, you will use the `_json_key`.

Examples:

- alexmorreale

- alex@pfc.com
- _json_key

hub.imagePullSecret.password

Password of the user you want to use to connect to your private registry.

Examples:

- plaintextpassword
- abc123SECRETzyx098

For gcr.io registries the password will be a big JSON blob for a Google cloud service account, it should look something like below.

```
password: |-
  {
    "type": "service_account",
    "project_id": "jupyter-se",
    "private_key_id": "f2ba09118a8d3123b3321bd9a7d6d0d9dc6fdb85",
    ...
  }
```

Learn more in this guide.

hub.image

Set custom image name / tag for the hub pod.

Use this to customize which hub image is used. Note that you must use a version of the hub image that was bundled with this particular version of the helm-chart - using other images might not work.

hub.image.name

Name of the image, without the tag.

```
# example names
yuvipanda/wikimedia-hub
gcr.io/my-project/my-hub
```

hub.image.tag

The tag of the image to pull.

This is the value after the `:` in your full image name.

```
# example tags
v1.11.1
zhy270a
```

hub.image.pullSecrets

Use an existing kubernetes secret to pull the custom image.

```yaml
# example existing pull secret.
singleuser:
  image:
    pullSecrets:
      - gcr-pull
```

hub.db

hub.db.type

Type of database backend to use for the hub database.

The Hub requires a persistent database to function, and this lets you specify where it should be stored.

The various options are:

1. **sqlite-pvc**

   Use an `sqlite` database kept on a persistent volume attached to the hub.

   By default, this disk is created by the cloud provider using *dynamic provisioning* configured by a storage class. You can customize how this disk is created / attached by setting various properties under `hub.db.pvc`.

   This is the default setting, and should work well for most cloud provider deployments.

2. **sqlite-memory**

   Use an in-memory `sqlite` database. This should only be used for testing, since the database is erased whenever the hub pod restarts - causing the hub to lose all memory of users who had logged in before.

   When using this for testing, make sure you delete all other objects that the hub has created (such as user pods, user PVCs, etc) every time the hub restarts. Otherwise you might run into errors about duplicate resources.

3. **mysql**

   Use an externally hosted mysql database.

   You have to specify an sqlalchemy connection string for the mysql database you want to connect to in `hub.db.url` if using this option.

   The general format of the connection string is:

   ```
   mysql+pymysql://<db-username>:<db-password>@<db-hostname>:<db-port>/<db-name>
   ```

   The user specified in the connection string must have the rights to create tables in the database specified.

   Note that if you use this, you *must* also set `hub.cookieSecret`.

4. **postgres**

   Use an externally hosted postgres database.

   You have to specify an sqlalchemy connection string for the postgres database you want to connect to in `hub.db.url` if using this option.

   The general format of the connection string is:

```
postgres+psycopg2://<db-username>:<db-password>@<db-hostname>:<db-port>/<db-name>
```

The user specified in the connection string must have the rights to create tables in the database specified.

Note that if you use this, you *must* also set `hub.cookieSecret`.

hub.db.pvc

Customize the Persistent Volume Claim used when `hub.db.type` is `sqlite-pvc`.

hub.db.pvc.annotations

Annotations to apply to the PVC containing the sqlite database.

See the Kubernetes documentation for more details about annotations.

hub.db.pvc.selector

Label selectors to set for the PVC containing the sqlite database.

Useful when you are using a specific PV, and want to bind to that and only that.

See the Kubernetes documentation for more details about using a label selector for what PV to bind to.

hub.db.pvc.storage

Size of disk to request for the database disk.

hub.db.url

Connection string when `hub.db.type` is mysql or postgres.

See documentation for `hub.db.type` for more details on the format of this property.

hub.db.password

Password for the database when `hub.db.type` is mysql or postgres.

hub.labels

Extra labels to add to the hub pod.

See the Kubernetes docs to learn more about labels.

hub.initContainers

list of initContainers to be run with hub pod. See Kubernetes Docs

```
hub:
  initContainers:
    - name: init-myservice
      image: busybox:1.28
      command: ['sh', '-c', 'command1']
    - name: init-mydb
      image: busybox:1.28
      command: ['sh', '-c', 'command2']
```

hub.extraEnv

Extra environment variables that should be set for the hub pod.

```
hub:
  extraEnv:
    MY_ENV_VARS_NAME: "my env vars value"
```

NOTE: We still support this field being a list of EnvVar objects as well.

These are usually used in two circumstances:

- Passing parameters to some custom code specified with `extraConfig`
- Passing parameters to an authenticator or spawner that can be directly customized by environment variables (rarer)

hub.extraConfig

Arbitrary extra python based configuration that should be in `jupyterhub_config.py`.

This is the *escape hatch* - if you want to configure JupyterHub to do something specific that is not present here as an option, you can write the raw Python to do it here.

extraConfig is a *dict*, so there can be multiple configuration snippets under different names. The configuration sections are run in alphabetical order.

Non-exhaustive examples of things you can do here:

- Subclass authenticator / spawner to do a custom thing
- Dynamically launch different images for different sets of images
- Inject an auth token from GitHub authenticator into user pod
- Anything else you can think of!

Since this is usually a multi-line string, you want to format it using YAML's | operator.

For example:

```
hub:
  extraConfig:
    myConfig.py: |
      c.JupyterHub.something = 'something'
      c.Spawner.somethingelse = 'something else'
```

No validation of this python is performed! If you make a mistake here, it will probably manifest as either the hub pod going into `Error` or `CrashLoopBackoff` states, or in some special cases, the hub running but… just doing very random things. Be careful!

hub.uid

The UID the hub process should be running as. Use this only if you are building your own image & know that a user with this uid exists inside the hub container! Advanced feature, handle with care! Defaults to 1000, which is the uid of the `jovyan` user that is present in the default hub image.

hub.fsGid

The gid the hub process should be using when touching any volumes mounted. Use this only if you are building your own image & know that a group with this gid exists inside the hub container! Advanced feature, handle with care! Defaults to 1000, which is the gid of the `jovyan` user that is present in the default hub image.

hub.service

Object to configure the service the JupyterHub will be exposed on by the Kubernetes server.

hub.service.type

The Kubernetes ServiceType to be used.

The default type is `ClusterIP`. See the Kubernetes docs to learn more about service types.

hub.service.loadBalancerIP

The public IP address the hub service should be exposed on.

This sets the IP address that should be used by the LoadBalancer for exposing the hub service. Set this if you want the hub service to be provided with a fixed external IP address instead of a dynamically acquired one. Useful to ensure a stable IP to access to the hub with, for example if you have reserved an IP address in your network to communicate with the JupyterHub.

To be provided like:

```
hub:
  service:
    loadBalancerIP: xxx.xxx.xxx.xxx
```

hub.service.loadBalancerSourceRanges

A list of IP CIDR ranges that are allowed to access the load balancer service. Defaults to allowing everyone to access it.

For more information please check the Kubernetes documentation.

hub.service.ports

Object to configure the ports the hub service will be deployed on.

hub.service.ports.nodePort

The nodePort to deploy the hub service on.

hub.service.annotations

Kubernetes annotations to apply to the hub service.

hub.pdb

Set the Pod Disruption Budget for the hub pod.

See the Kubernetes documentation for more details about disruptions.

hub.pdb.enabled

Whether PodDisruptionBudget is enabled for the hub pod.

hub.pdb.minAvailable

Minimum number of pods to be available during the voluntary disruptions.

hub.existingSecret

Name of the existing secret in the kubernetes cluster, typically the `hub-secret`.

This secret should represent the structure as otherwise generated by this chart:

```yaml
apiVersion: v1
data:
  proxy.token: < FILL IN >
  values.yaml: < FILL IN >
kind: Secret
metadata:
  name: hub-secret
```

NOTE: if you choose to manage the secret yourself, you are in charge of ensuring the secret having the proper contents.

proxy

proxy.secretToken

A 32-byte cryptographically secure randomly generated string used to secure communications between the hub and the configurable-http-proxy.

```
# to generate a value, run
openssl rand -hex 32
```

Changing this value will cause the proxy and hub pods to restart. It is good security practice to rotate these values over time. If this secret leaks, *immediately* change it to something else, or user data can be compromised

proxy.service

Object to configure the service the JupyterHub's proxy will be exposed on by the Kubernetes server.

proxy.service.type

See `hub.service.type`.

proxy.service.labels

Extra labels to add to the proxy service.

See the Kubernetes docs to learn more about labels.

proxy.service.annotations

Annotations to apply to the service that is exposing the proxy.

See the Kubernetes documentation for more details about annotations.

proxy.service.nodePorts

Object to set NodePorts to expose the service on for http and https.

See the Kubernetes documentation for more details about NodePorts.

proxy.service.nodePorts.http

The HTTP port the proxy-public service should be exposed on.

proxy.service.nodePorts.https

The HTTPS port the proxy-public service should be exposed on.

proxy.service.loadBalancerIP

See `hub.service.loadBalancerIP`

proxy.https

Object for customizing the settings for HTTPS used by the JupyterHub's proxy. For more information on configuring HTTPS for your JupyterHub, see the HTTPS section in our security guide

proxy.https.enabled

Indicator to set whether HTTPS should be enabled or not on the proxy. Defaults to `true` if the https object is provided.

proxy.https.type

The type of HTTPS encryption that is used. Decides on which ports and network policies are used for communication via HTTPS. Setting this to `secret` sets the type to manual HTTPS with a secret that has to be provided in the `https.secret` object. Defaults to `letsencrypt`.

proxy.https.letsencrypt

proxy.https.letsencrypt.contactEmail

The contact email to be used for automatically provisioned HTTPS certificates by Let's Encrypt. For more information see Set up automatic HTTPS. Required for automatic HTTPS.

proxy.https.manual

Object for providing own certificates for manual HTTPS configuration. To be provided when setting `https.type` to `manual`. See Set up manual HTTPS

proxy.https.manual.key

The RSA private key to be used for HTTPS. To be provided in the form of

```
key: |
  -----BEGIN RSA PRIVATE KEY-----
  ...
  -----END RSA PRIVATE KEY-----
```

proxy.https.manual.cert

The certificate to be used for HTTPS. To be provided in the form of

```
cert: |
  -----BEGIN CERTIFICATE-----
  ...
  -----END CERTIFICATE-----
```

proxy.https.secret

Secret to be provided when setting `https.type` to `secret`.

proxy.https.secret.name

Name of the secret

proxy.https.secret.key

Path to the private key to be used for HTTPS. Example: `'tls.key'`

proxy.https.secret.crt

Path to the certificate to be used for HTTPS. Example: `'tls.crt'`

proxy.https.hosts

You domain in list form. Required for automatic HTTPS. See Set up automatic HTTPS. To be provided like:

```
hosts:
  - <your-domain-name>
```

proxy.pdb

Set the Pod Disruption Budget for the proxy pod.

See the Kubernetes documentation for more details about disruptions.

proxy.pdb.enabled

Whether PodDisruptionBudget is enabled for the proxy pod.

proxy.pdb.minAvailable

Minimum number of pods to be available during the voluntary disruptions.

auth

auth.state

auth.state.enabled

Enable persisting auth_state (if available). See: http://jupyterhub.readthedocs.io/en/latest/api/auth.html

auth.state.cryptoKey

auth_state will be encrypted and stored in the Hub's database. This can include things like authentication tokens, etc. to be passed to Spawners as environment variables. Encrypting auth_state requires the cryptography package. It must contain one (or more, separated by ;) 32-byte encryption keys. These can be either base64 or hex-encoded. The JUPYTERHUB_CRYPT_KEY environment variable for the hub pod is set using this entry.

```
# to generate a value, run
openssl rand -hex 32
```

If encryption is unavailable, auth_state cannot be persisted.

singleuser

Options for customizing the environment that is provided to the users after they log in.

singleuser.cpu

Set CPU limits & guarantees that are enforced for each user. See: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

singleuser.cpu.limit

singleuser.cpu.guarantee

singleuser.memory

Set Memory limits & guarantees that are enforced for each user.

See the Kubernetes docs for more info.

singleuser.memory.limit

singleuser.memory.guarantee

Note that this field is referred to as *requests* by the Kubernetes API.

singleuser.imagePullSecret

Creates an image pull secret for you and makes the user pods utilize it, allowing them to pull images from private image registries.

Using this configuration option automates the following steps that normally is required to pull from private image registries.

```
# you won't need to run this manually...
kubectl create secret docker-registry singleuser-image-credentials \
  --docker-server=<REGISTRY> \
  --docker-username=<USERNAME> \
```

(continues on next page)

(continued from previous page)

```
  --docker-email=<EMAIL> \
  --docker-password=<PASSWORD>
```

```
# you won't need to specify this manually...
spec:
  imagePullSecrets:
    - name: singleuser-image-credentials
```

To learn the username and password fields to access a gcr.io registry from a Kubernetes cluster not associated with the same google cloud credentials, look into this guide and read the notes about the password.

singleuser.imagePullSecret.enabled

Enable the creation of a Kubernetes Secret containing credentials to access a image registry. By enabling this, user pods and image puller pods will also be configured to use these credentials when they pull their container images.

singleuser.imagePullSecret.registry

Name of the private registry you want to create a credential set for. It will default to Docker Hub's image registry.

Examples:

- https://index.docker.io/v1/
- quay.io
- eu.gcr.io
- alexmorreale.privatereg.net

singleuser.imagePullSecret.username

Name of the user you want to use to connect to your private registry. For external gcr.io, you will use the `_json_key`.

Examples:

- alexmorreale
- alex@pfc.com
- _json_key

singleuser.imagePullSecret.password

Password of the user you want to use to connect to your private registry.

Examples:

- plaintextpassword
- abc123SECRETzyx098

For gcr.io registries the password will be a big JSON blob for a Google cloud service account, it should look something like below.

```
password: |-
  {
    "type": "service_account",
    "project_id": "jupyter-se",
    "private_key_id": "f2ba09118a8d3123b3321bd9a7d6d0d9dc6fdb85",
    ...
  }
```

Learn more in this guide.

singleuser.image

Set custom image name / tag used for spawned users.

This image is used to launch the pod for each user.

singleuser.image.name

Name of the image, without the tag.

Examples:

- yuvipanda/wikimedia-hub-user
- gcr.io/my-project/my-user-image

singleuser.image.tag

The tag of the image to use.

This is the value after the `:` in your full image name.

singleuser.image.pullPolicy

Set the imagePullPolicy on the singleuser pods that are spun up by the hub.

See the Kubernetes docs for more info.

singleuser.image.pullSecrets

Use an existing kubernetes secret to pull the custom image.

```
# example existing pull secret
singleuser:
  image:
    pullSecrets:
      - gcr-pull
```

singleuser.profileList

For more information about the profile list, see KubeSpawner’s documentation as this is simply a passthrough to that configuration.

NOTE: The image-pullers are aware of the overrides of images in `singleuser.profileList` but they won’t be if you configure it in JupyterHub’s configuration of ‘`c.KubeSpawner.profile_list`.

```yaml
singleuser:
  profileList:
    - display_name: "Default: Shared, 8 CPU cores"
      description: "Your code will run on a shared machine with CPU only."
      default: True
    - display_name: "Personal, 4 CPU cores & 26GB RAM, 1 NVIDIA Tesla K80 GPU"
      description: "Your code will run a personal machine with a GPU."
      kubespawner_override:
        extra_resource_limits:
          nvidia.com/gpu: "1"
```

singleuser.schedulerStrategy

Deprecated and no longer does anything. Use the user-scheduler instead in order to accomplish a good packing of the user pods.

singleuser.extraTolerations

Tolerations allow a pod to be scheduled on nodes with taints. These are additional tolerations other than the user pods and core pods default ones `hub.jupyter.org/dedicated=user:NoSchedule` or `hub.jupyter.org/dedicated=core:NoSchedule`. Note that a duplicate set of tolerations exist where `/` is replaced with `_` as the Google cloud does not support the character `/` yet in the toleration.

See the Kubernetes docs for more info.

Pass this field an array of `Toleration` objects.

singleuser.extraNodeAffinity

Affinities describe where pods prefer or require to be scheduled, they may prefer or require a node where they are to be scheduled to have a certain label (node affinity). They may also require to be scheduled in proximity or with a lack of proximity to another pod (pod affinity and anti pod affinity).

See the Kubernetes docs for more info.

singleuser.extraNodeAffinity.required

Pass this field an array of `NodeSelectorTerm` objects.

singleuser.extraNodeAffinity.preferred

Pass this field an array of `PreferredSchedulingTerm` objects.

singleuser.extraPodAffinity

See the description of `singleuser.extraNodeAffinity`.

singleuser.extraPodAffinity.required

Pass this field an array of `PodAffinityTerm` objects.

singleuser.extraPodAffinity.preferred

Pass this field an array of `WeightedPodAffinityTerm` objects.

singleuser.extraPodAntiAffinity

See the description of `singleuser.extraNodeAffinity`.

singleuser.extraPodAntiAffinity.required

Pass this field an array of `PodAffinityTerm` objects.

singleuser.extraPodAntiAffinity.preferred

Pass this field an array of `WeightedPodAffinityTerm` objects.

scheduling

Objects for customizing the scheduling of various pods on the nodes and related labels.

scheduling.userScheduler

The user scheduler is making sure that user pods are scheduled tight on nodes, this is useful for autoscaling of user node pools.

scheduling.userScheduler.enabled

Enables the user scheduler.

scheduling.userScheduler.replicas

You can have multiple schedulers to share the workload or improve availability on node failure.

scheduling.userScheduler.image

The image containing the kube-scheduler binary.

scheduling.userScheduler.image.name

scheduling.userScheduler.image.tag

scheduling.userScheduler.pdb

Set the Pod Disruption Budget for the user scheduler.

See the Kubernetes documentation for more details about disruptions.

scheduling.userScheduler.pdb.enabled

Whether PodDisruptionBudget is enabled for the user scheduler.

scheduling.userScheduler.pdb.minAvailable

Minimum number of pods to be available during the voluntary disruptions.

scheduling.podPriority

Pod Priority is used to allow real users evict placeholder pods that in turn triggers a scale up by a cluster autoscaler. So, enabling this option will only make sense if the following conditions are met:

1. Your Kubernetes cluster has at least version 1.11
2. A cluster autoscaler is installed
3. user-placeholer pods is configured to get a priority equal or higher than the cluster autoscaler's priority cutoff
4. Normal user pods have a higher priority than the user-placeholder pods

Note that if the default priority cutoff if not configured on cluster autoscaler, it will currently default to 0, and that in the future this is meant to be lowered. If your cloud provider is installing the cluster autoscaler for you, they may also configure this specifically.

Recommended settings for a cluster autoscaler…

… with a priority cutoff of -10 (GKE):

```yaml
podPriority:
  enabled: true
  globalDefault: false
  defaultPriority: 0
  userPlaceholderPriority: -10
```

… with a priority cutoff of 0:

```yaml
podPriority:
  enabled: true
  globalDefault: true
  defaultPriority: 10
  userPlaceholderPriority: 0
```

scheduling.podPriority.enabled

scheduling.podPriority.globalDefault

Warning! This will influence all pods in the cluster.

The priority a pod usually get is 0. But this can be overridden with a PriorityClass resource if it is declared to be the global default. This configuration option allows for the creation of such global default.

scheduling.podPriority.defaultPriority

The actual value for the default pod priority.

scheduling.podPriority.userPlaceholderPriority

The actual value for the user-placeholder pods' priority.

scheduling.userPlaceholder

User placeholders simulate users but will thanks to PodPriority be evicted by the cluster autoscaler if a real user shows up. In this way placeholders allow you to create a headroom for the real users and reduce the risk of a user having to wait for a node to be added. Be sure to use the the continuous image puller as well along with placeholders, so the images are also available when real users arrive.

To test your setup efficiently, you can adjust the amount of user placeholders with the following command:

```
# Configure to have 3 user placeholders
kubectl scale sts/user-placeholder --replicas=3
```

scheduling.userPlaceholder.enabled

scheduling.userPlaceholder.replicas

How many placeholder pods would you like to have?

scheduling.userPlaceholder.resources

Unless specified here, the placeholder pods will request the same resources specified for the real singleuser pods.

scheduling.corePods

These settings influence the core pods like the hub, proxy and user-scheduler pods.

scheduling.corePods.nodeAffinity

Where should pods be scheduled? Perhaps on nodes with a certain label is preferred or even required?

scheduling.corePods.nodeAffinity.matchNodePurpose

Decide if core pods *ignore*, *prefer* or *require* to schedule on nodes with this label:

```
hub.jupyter.org/node-purpose=core
```

scheduling.userPods

These settings influence the user pods like the user-placeholder, user-dummy and actual user pods named like jupyter-someusername.

scheduling.userPods.nodeAffinity

Where should pods be scheduled? Perhaps on nodes with a certain label is preferred or even required?

scheduling.userPods.nodeAffinity.matchNodePurpose

Decide if user pods *ignore*, *prefer* or *require* to schedule on nodes with this label:

```
hub.jupyter.org/node-purpose=user
```

ingress

ingress.enabled

Enable the creation of a Kubernetes Ingress to proxy-public service.

See Advanced Topics — Zero to JupyterHub with Kubernetes 0.7.0 documentation for more details.

ingress.annotations

Annotations to apply to the Ingress.

See the Kubernetes documentation for more details about annotations.

ingress.hosts

List of hosts to route requests to the proxy.

ingress.pathSuffix

Suffix added to Ingress's routing path pattern.

Specify `*` if your ingress matches path by glob pattern.

ingress.tls

TLS configurations for Ingress.

See the Kubernetes documentation for more details about annotations.

prePuller

prePuller.hook

See the *optimization section* for more details.

prePuller.hook.enabled

prePuller.continuous

See the *optimization section* for more details.

NOTE: If used with a Cluster Autoscaler (an autoscaling node pool), also add user-placeholders and enable pod priority.

prePuller.continuous.enabled

prePuller.extraImages

See the *optimization section* for more details.

```
prePuller:
  extraImages:
    myExtraImageIWantPulled:
      name: jupyter/all-spark-notebook
      tag: 2343e33dec46
```

custom

Additional values to pass to the Hub. JupyterHub will not itself look at these, but you can read values in your own custom config via `hub.extraConfig`. For example:

```
custom:
  myHost: "https://example.horse"
hub:
  extraConfig:
    myConfig.py: |
      c.MyAuthenticator.host = get_config("custom.myHost")
```

7.1.2 Related Projects

- The JupyterHub Documentation provides information about JupyterHub itself (not the Kubernetes deployment).
- Binder allows users to create sharable computational environments on-the-fly. It makes heavy use of JupyterHub.

- The 2016 JupyterHub Workshop was an informal gathering to share experience in deploying JupyterHub for various use-cases, including teaching and high-performance computing.

7.1.3 Utilized Tools

JupyterHub is meant to connect with many tools in the world of cloud computing and container technology. This page describes these tools in greater detail in order to provide some more contextual information.

Cloud Computing Providers

This is whatever will run the actual computation. Generally it means a company, university server, or some other organization that hosts computational resources that can be accessed remotely. JupyterHub will run on these computational resources, meaning that users will also be operating on these resources if they're interacting with your JupyterHub.

They provide the following things:

- Computing
- Disk space
- Networking (both internal and external)
- Creating, resizing, and deleting clusters

Some of these organizations are companies (e.g., Google), though JupyterHub will work fine with university clusters or custom cluster deployments as well. For these materials, any cluster with Kubernetes installed will work with JupyterHub.

More information about setting up accounts services with cloud providers can be found here.

Container Technology

Container technology is essentially the idea of bundling all of the necessary components to run a piece of software. There are many ways to do this, but one that we'll focus on is called Docker. Here are the main concepts of Docker:

Container Image

Container images contain the dependencies required to run your code. This includes **everything**, all the way down to the operating system itself. It also includes things like the filesystem on which your code runs, which might include data etc. Containers are also portable, meaning that you can exactly recreate the computational environment to run your code on almost any machine.

In Docker, images are described as layers, as in layers of dependencies. For example, say you want to build a container that runs scikit-learn. This has a dependency on Python, so you have two layers: one for python, and another that inherits the python layer and adds the extra piece of scikit-learn. Moreover, that base python layer needs an operating system to run on, so now you have three layers: ubuntu -> python -> scikit-learn. You get the idea. The beauty of this is that it means you can share base layers between images. This means that if you have many different images that all require ubuntu, you don't need to have many copies of ubuntu lying around.

Images can be created from many things. If you're using Docker, the basic way to do this is with a **Dockerfile**. This is essentially a list of instructions that tells Docker how to create an image. It might tell Docker which base layers you want to include in an image, as well as some extra dependencies that you need in the image. Think of it like a recipe that tells Docker how to create an image.

Containers

You can "run" a container image, and it creates a container for you. A container is a particular instantiation of a container image. This means that it actually exists on a computer. It is a self-contained computational environment that is constructed according to the layers that are inside of the Container Image. However, because it is now running on the computer, it can do other useful things like talk to other Docker containers or communicate via the internet.

Kubernetes

Kubernetes is a service that runs on cloud infrastructures. It provides a single point of contact with the machinery of your cluster deployment, and allows a user to specify the computational requirements that they need (e.g., how many machines, how many CPUs per machine, how much RAM). Then, it handles the resources on the cluster and ensures that these resources are always available. If something goes down, Kubernetes will try to automatically bring it back up.

Kubernetes can only manage the computing resources that it is given. This means that it generally can **not** create new resources on its own (with the exception of disk space).

The following sections describe some objects in Kubernetes that are most relevant for JupyterHub.

Processes

Are any program that is running on a machine. For example, a Jupyter Notebook creates several processes that handle the execution of code and the display in the browser. This isn't technically a Kubernetes object, since literally any computer has processes that run on it, but Kubernetes does keep track of running processes in order to ensure that they remain running if needed.

Pods

Pods are essentially a collection of one or more *containers* that run together. You can think of them as a way of combining containers that, as a group, accomplish some goal.

For example, say you want to create a web server that is open to the world, but you also want authentication so that only a select group of users can access it. You could use a single pod with two containers.

- One that does the authentication. It would have something like Apache specified in its container image, and would be connected to the outside world.
- One that receives information from the authentication container, and does something fancy with it (maybe it runs a python process).

This is useful because it lets you compartmentalize the components of the service that you want to run, which makes things easier to manage and keeps things more stable.

For more information about pods, see the Kubernetes documentation about pods.

Deployments

A deployment is a collection of pods on Kubernetes. It is how Kubernetes knows exactly what containers and what machines need to be running at all times. For example, if you have two pods: one that does the authenticating described above, and another that manages a database, you can specify both in a deployment.

Kubernetes will ensure that both pods are active, and if one goes down then it will try to re-create it. It does this by continually checking the current state of the pods, and then comparing this with the original specification of the deployment. If there are differences between the current state vs. the specification of the deployment, Kubernetes will attempt to make changes until the current state matches the specification.

For more information about deployments, see the Kubernetes documentation about deployment.

Note: Users don't generally "create" deployments directly, they are instead generated from a set of instructions that are sent to Kubernetes. We'll cover this in the section on "Helm".

Service

A service is simply a stable way of referring to a deployment. Kubernetes is all about intelligently handling dynamic and quickly-changing computational environments. This means that the VMs running your pods may change, IP addresses will be different, etc. However you don't want to have to re-orient yourself every time this happens. A Kubernetes service keeps track of all these changes on the backend, and provides a single address to manage your deployment.

For more information about services, see the Kubernetes documentation about services.

Namespace

Finally, a namespace defines a collection of objects in Kubernetes. It is generally the most "high-level" of the groups we've discussed thus far. For example, a namespace could be a single class running with JupyterHub.

For more information about namespaces, see the Kubernetes documentation on namespaces.

Persistent Volume Claim

Persistent Volume Claims are a way to have persistent storage without being tied down to one specific computer or machine. Kubernetes is about that flexibility, and that means that we don't want to lock ourselves into a particular operating system just because our files are already on it. Persistent Volume Claims help deal with this problem by knowing how to convert files between disk types (e.g., AWS vs. Google disks).

For more information on Persistent Volume Claims, see the Kubernetes documentation on persistent volumes.

Helm

Helm is a way of specifying Kubernetes objects with a standard template.

Charts

The way that Helm controls Kubernetes is with templates of structured information that specify some computational requirements. These templates are called "charts", or "helm charts". They contain all of the necessary information for Kubernetes to generate:

- a deployment object
- a service object
- a persistent volume object for a deployment.
- collections of the above components

They can be installed into a namespace, which causes Kubernetes to begin deploying the objects above into that namespace.

Charts have both names and versions, which means that you can easily update them and build off of them. There are community maintained charts available, and we use a chart to install and upgrade JupyterHub in this guide. In our case, the helm chart is a file called `config.yaml`.

Releases

A release is basically a specific instantiation of a helmchart inserted into a particular namespace. If you'd like to upgrade your kubernetes deployment (say, by changing the amount of RAM that each user should get), then you can change the helm chart, then re-deploy it to your Kubernetes cluster. This generates a new version of the release.

JupyterHub

JupyterHub is a way of utilizing the components above in order to provide computational environments that users can access remotely. It exists as two Kubernetes deployments, Proxy and Hub, each of which has one pod. Each deployment accomplishes some task that, together, make up JupyterHub. Finally, the output of JupyterHub is a user pod, which specifies the computational environment in which a single user will operate. So essentially a JupyterHub is a collection of:

- Pods that contain the JupyterHub Machinery
- A bunch of user pods that are constantly being created or destroyed.

Below we'll describe the primary JupyterHub pods.

Proxy Pod

This is the user-facing pod. It provides the IP address that people will go to in order to access JupyterHub. When a new users goes to this pod, it will decide whether to:

- send that user to the Hub pod, which will create a container for that user, or
- if that user's container already exists, send them directly to that container instead.

Information about the user's identity is stored as a cookie on their computer. This is how the proxy pod knows whether a user already has a running container.

Hub Pod

Receives traffic from the proxy pod. It has 3 main running processes:

1. An authenticator, which can verify a user's account. It also contains a process.
2. A "KubeSpawner" that talks to the Kubernetes API and tells it to spawn pods for users if one doesn't already exist. KubeSpawner will tell Kubernetes to create a pod for a new user, then it will tell the Proxy Pod that the user's pod has been created.
3. An admin panel that has information about who has pods created, and what kind of usage exists on the cluster.

7.1.4 Glossary

A partial glossary of terms used in this guide. For more complete descriptions of the components in JupyterHub, see *Utilized Tools*. Here we try to keep the definition as succinct and relevant as possible, and provide links to learn more details.

admin user A user who can access the JupyterHub admin panel. They can start/stop user pods, and potentially access their notebooks.

authenticator The way in which users are authenticated to log into JupyterHub. There are many authenticators available, like GitHub, Google, MediaWiki, Dummy (anyone can log in), etc.

`config.yaml` The *Helm charts* templates are rendered with these *Helm values* as input. The file is written in the YAML format. The YAML format is essential to grasp if working with Kubernetes and Helm.

container A container is a isolated working space which for us gives users the tools, libraries, and capabilities to be productive.

culler A separate process in the JupyterHub that stops the user pods of users who have not been active in a configured interval.

Dockerfile A Dockerfile declares how to build a *Docker image*.

Docker image A Docker image, built from a *Dockerfile*, allows tools like `docker` to create any number of *containers*.

image registry A service for storing Docker images so that they can be stored and used later. The default public registry is at https://hub.docker.com, but you can also run your own private image registry. Many cloud providers offer private image registry services.

environment variables A set of named values that can affect the way running processes will behave on a computer. Some common examples are `PATH`, `HOME`, and `EDITOR`.

Helm chart A Helm chart is a group of *Helm templates* that can, given its default values and overrides in provided `yaml` files, render to a set of *Kubernetes resources* that can be easily installed to your Kubernetes cluster. In other words a Helm chart is like a configurable installation of software and infrastructure to exist on a cloud.

Helm template A Helm template (`.yaml` files), can given values, render to a *Kubernetes resource*.

Helm values *Helm charts* has a set of predefined values (`values.yaml`) typically overridden by other values in `config.yaml`. The final values are used to generate *Kubernetes resources* from *Helm templates* within a *Helm chart*.

Kubernetes For our purposes, you can think of Kubernetes as a way to speak to a cloud and describe what you would like it to do, in a manner that isn't specific for that cloud.

- The Illustrated Children's Guide to Kubernetes
- The official "What is Kubernetes?" text

Kubernetes API server The Kubernetes API server, also referred to as the master, will answer questions and update the desired state of the cluster for you. When you use `kubectl` you communicate with the API server.

Kubernetes Pod *Pods* are the smallest deployable units of computing that can be created and managed in Kubernetes. A pod will use a *Docker image* to create a container, and most often a controller such as a Deployment will ensure there is always X running pods of a kind.

See the Kubernetes documentation for more information.

Kubernetes resource A Kubernetes resource can for example be a Deployment, Service or a Secret. It is something you can request by the *Kubernetes API server* to be present in the cluster.

persistent storage A filesystem attached to a user pod that allows the user to store notebooks and files that persist across multiple logins.

Node Pool A *node pool* or *node group* represents a set of nodes of the same kind. With cluster autoscaling, a node pool can grow and shrink based on demand allowing you to save computational resources.

repo2docker A tool which lets you quickly convert a Git repository into a *Docker image*.

spawner A spawner is a separate process created for each active user by JupyterHub. They are each responsible for one user. This Helm chart relies on KubeSpawner.

CHAPTER 8

Institutional support

This guide and the associated helm chart would not be possible without the amazing institutional support from the following organizations (and the organizations that support them!)

- UC Berkeley Data Science Division
- Berkeley Institute for Data Science
- Cal Poly, San Luis Obispo
- Simula Research Institute

Index

A

admin user, 115
authenticator, 115

C

config.yaml, 115
container, 115
culler, 115

D

Docker image, 115
Dockerfile, 115

E

environment variables, 115

H

Helm chart, 115
Helm template, 115
Helm values, 115

I

image registry, 115

K

Kubernetes, 115
Kubernetes API server, 115
Kubernetes Pod, 115
Kubernetes resource, 115

N

Node Pool, 116

P

persistent storage, 115

R

repo2docker, 116

S

spawner, 116