Zero to JupyterHub

Release 0.4

Chris Holdgraf

Aug 16, 2018

JupyterHub is a tool that allows you to quickly utilize cloud computing infrastructure to manage a hub that enables your users to interact remotely with a computing environment that you specify. JupyterHub offers a useful way to standardize the computing environment of a group of people (e.g., for a class of students or an analytics team), as well as allowing people to access the hub remotely.

This growing collection of information will help you set up your own JupyterHub instance. It is in an early stage, so the information and tools may change quickly.

If you have tips or deployments that you would like to share, see *Resources from the community*. If you see anything that is incorrect or have any questions, feel free to reach out at the issues page.

CHAPTER 1

Getting to Step Zero: your Kubernetes cluster

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

1.1 Creating a Kubernetes Cluster

Kubernetes' documentation describes the many ways to set up a cluster. Here, we shall provide quick instructions for the most painless and popular ways of getting setup in various cloud providers and on other infrastructure:

- *Google Cloud*
- *Microsoft Azure*
- *Amazon AWS*
- *Red Hat OpenShift*

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.

- If you are concerned at all about security (you probably should be), see the Kubernetes best-practices guide for information about keeping your Kubernetes infrastructure secure.

1.2 Step Zero: Kubernetes on Google Cloud

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 `https://console.cloud.google.com` and log in.

---

2. Enable the Kubernetes Engine API.

3. Use your preferred command line interface.

   You have two options: a) use the Google Cloud Shell (no installation needed) or b) install and use the gcloud command-line tool. If you are unsure which to choose, we recommend beginning with option "a" and using the Google Cloud Shell. Instructions for each are detailed below:

   (a) **Use the Google Cloud Shell**. Start the Google Cloud Shell by clicking the button shown below. This will start an interactive shell session within Google Cloud.
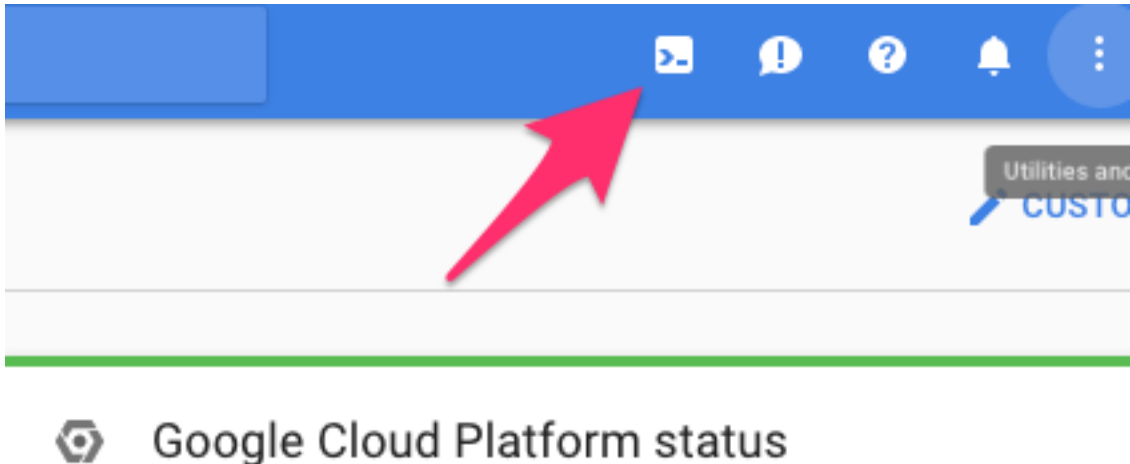
   See the Google Cloud Shell docs for more information.

   (b) **Install and use the gcloud command line tool**. This tool sends commands to Google Cloud and lets you do things like create and delete clusters.

   • Go to the gcloud command line tool downloads page to **download and install the gcloud command line tool**.

   • See the gcloud documentation for more information on the gcloud command line tool.

4. Install `kubectl`, which is a tool for controlling kubernetes. From the terminal, enter:

   ```
   gcloud components install kubectl
   ```

5. Create a Kubernetes cluster on Google Cloud, by typing the following command into either the Google Cloud shell or the gcloud command-line tool:

   ```
   gcloud container clusters create <YOUR-CLUSTER> \
       --num-nodes=3 \
       --machine-type=n1-standard-2 \
       --zone=us-central1-b
   ```

   where:

   • `--num-nodes` specifies how many computers to spin up. The higher the number, the greater the cost.

   • `--machine-type` specifies the amount of CPU and RAM in each node. There is a variety of types to choose from. Picking something appropriate here will have a large effect on how much you pay - smaller machines restrict the max amount of RAM each user can have access to but allow more fine-grained scaling, reducing cost. The default (`n1-standard-2`) has 2CPUs and 7.5G of RAM each, and might not be a good fit for all use cases!

- `--zone` specifies which data center to use. Pick something that is not too far away from your users. You can find a list of them here.

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.

---

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

```
kubectl get node
```

The response should list three running nodes.

7. Give your account super-user permissions, allowing you to perform all the actions needed to set up JupyterHub.

```
kubectl create clusterrolebinding cluster-admin-binding \
    --clusterrole=cluster-admin \
    --user=<YOUR-EMAIL-ADDRESS>
```

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

1.3 Step Zero: Kubernetes on Microsoft Azure Container Service (AKS)

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.
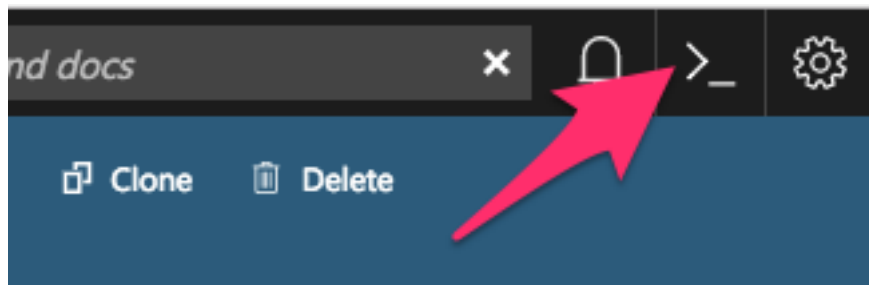
Note:

- 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.

```
az account set -s <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. Enable the cloud APIs required before creating a cluster.

The following commands enable various Azure tools that we'll need in creating and managing the JupyterHub.

```
az provider register --name Microsoft.Network --wait
az provider register --name Microsoft.Compute --wait
az provider register --name Microsoft.Storage --wait
az provider register --name Microsoft.ContainerService --wait
```

Note: Each of these commands may take up to several minutes to complete.

---

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 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 \
              --kubernetes-version 1.8.2 \
              --output table
```

where:

- `--name` is the name you want to use to refer to your cluster
- `--resource-group` is the ResourceGroup you created in step 4
- `--ssh-key-value` is the ssh public key created in step 7
- `--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.
- `--kubernetes-version` is the version of Kubernetes we want to use.

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
```

1.3. Step Zero: Kubernetes on Microsoft Azure Container Service (AKS)   **7**

where:

- `--name` is the name you gave your cluster in step 7
- `--resource-group` is the ResourceGroup you created in step 4

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: Azure AKS is still in **preview**, and not all features might work as intended. In particular,

1. You have to not use RBAC, since AKS does not support it yet.

2. You should skip step 2 (granting RBAC rights) with the "initialization" section *when setting up helm*.

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

1.4 Step Zero: 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.

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.

3. Install kops and kubectl on your CI host

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

4. Setup an ssh keypair to use with the cluster

   ```
   ssh-keygen
   ```

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. 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>
   ```

7. 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}'`
   ```

8. Set the availability zones for the nodes

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

   ```
   export ZONES=$(aws ec2 describe-availability-zones --region $REGION | grep
   →ZoneName | awk '{print $2}' | tr -d '"')
   export ZONES=$(echo $ZONES | tr -d " " | rev | cut -c 2- | rev)
   ```

9. 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)
   ```

   (continues on next page)

```
--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.

---

10. 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`

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

    Run:

    ```
    kubectl get nodes
    ```

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

    If you want to run kubectl from a box not on AWS, you can use run the following on AWS: `kops export kubecfg`

    To use kubctl and helm from a local machine, 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>`

12. Configure ssh bastion

    Skip this step if you did not go with the 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

13. 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.

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

1.5 JupyterHub 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).

1.5.1 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.

Creating your 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.

To begin, go to *Setting up Helm*.

2.1 Getting started with JupyterHub

JupyterHub lets you create custom computing environments that can be accessed remotely (e.g., at a specific URL) by multiple users.

This guide acts as an assistant to guide you through the process of setting up your JupyterHub deployment using Kubernetes. It helps you connect and configure the following things:

- A **cloud provider** such Google Cloud, Microsoft Azure, Amazon EC2, and others
- **Kubernetes** to manage resources on the cloud
- **Helm** to configure and control Kubernetes
- **Docker** to use containers that standardize computing environments
- **JupyterHub** to manage users and deploy Jupyter notebooks

You already are well on your way to understanding what it means (procedurally) to deploy Jupyterhub.

2.1.1 Verifying JupyterHub dependencies

At this point, you should have completed *Step Zero* and have an operational Kubernetes cluster. You will already have a cloud provider/infrastructure and kubernetes and docker installed.

If you need to create a Kubernetes cluster, see *Creating a Kubernetes Cluster*.

We also depend on Helm and the JupyterHub Helm chart for your JupyterHub deployment. We'll deploy them in this section. Let's begin by moving on to *Setting up Helm*.

Note: For a more extensive description of the tools and services that JupyterHub depends upon, see our *Tools used in a JupyterHub Deployment* page.

2.2 Setting up Helm

Helm, the package manager for Kubernetes, is a useful tool to install, upgrade and manage applications on a Kubernetes cluster. We will be using Helm to install and manage JupyterHub on our cluster.

Helm works by initializing itself both locally (on your computer) and remotely (on your kubernetes cluster). When you run `helm` commands, your local helm client sends instructions to the `Tiller`, which exists on your Kubernetes cluster, and is controlled by the server-side `helm` install.

2.2.1 Installation

The simplest way to install helm is to run Helm's installer script at a terminal:

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

Alternative methods for helm installation exist if you prefer to install without using the script.

2.2.2 Initialization

After installing helm on your machine, initialize helm on your Kubernetes cluster. At a terminal for your local machine (or within an interactive cloud shell from your provider), enter:

1. Set up a ServiceAccount for use by `Tiller`, the server side component of `helm`.

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

   Azure AKS: If you're on Azure AKS, you should now skip directly to step 3.**

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

   While most clusters have RBAC enabled and you need this line, you **must** skip this step if your kubernetes cluster does not have RBAC enabled (for example, if you are using Azure AKS).

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

3. Set up Helm on the cluster.

```
helm init --service-account tiller
```

This command only needs to run once per Kubernetes cluster.

Note: The local and remote version of `helm` must be the same in order to ensure they can talk to each other. If you wish to run `helm` commands from a *new* computer than the one used to run the commands above, you must re-initialize it by running the following modified version of the `init` command:

```
helm init --client-only --service-account tiller
```

This will initialize `helm` locally, according to the version that is running remotely on the cluster. Note that this requires `kubectl` to point to the correct kubernetes cluster. See the kubernetes context manager for more details.

2.2.3 Verify

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

```
helm version
```

It should provide output like:

```
Client: &version.Version{SemVer:"v2.8.1", GitCommit:
→"46d9ea82e2c925186e1fc620a8320ce1314cbb02", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.8.1", GitCommit:
→"46d9ea82e2c925186e1fc620a8320ce1314cbb02", GitTreeState:"clean"}
```

Make sure you have at least version 2.8.1!

If you receive an error that the Server is unreachable, do another `helm version` in 15-30 seconds, and it should display the Server version.

Secure Helm

Ensure that tiller is secure from access inside the cluster:

```
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"]}]'
```

2.2.4 Next Step

Congratulations. Helm is now set up. The next step is to *install JupyterHub*!

2.3 Setting up JupyterHub

Now that we have a Kubernetes cluster and helm setup, we can begin setting up a JupyterHub.

2.3.1 Prepare configuration file

This step prepares a configuration file (config file). We will use the YAML file format to specify JupyterHub's configuration.

It's important to save the config file in a safe place. The config file is needed for future changes to JupyterHub's settings.

For the following steps, use your favorite code editor. We'll use the nano editor as an example.

1. Create a file called `config.yaml`. Using the nano editor, for example, entering `nano config.yaml` at the terminal will start the editor and open the config file.

2. Create a random hex string to use as a security token. Run this command in a terminal

```
openssl rand -hex 32
```

Copy the output for use in the next step

3. Insert these lines into the `config.yaml` file. When editing YAML files, use straight quotes and spaces and avoid using curly quotes or tabs. Substitute `RANDOM_STRING` below with the output of `openssl rand -hex 32` from step 2.

```
proxy:
  secretToken: "<OUTPUT-OF-`openssl rand -hex 32`>"
```

4. **Azure AKS only** If you're on Microsoft Azure AKS, you must disable RBAC. Do so by putting the following in `config.yaml`

```
rbac:
    enabled: false
```

See the RBAC documentation for more details.

5. Save the `config.yaml` file.

2.3.2 Install JupyterHub

1. Let's add the JupyterHub helm repository to your helm, so you can install JupyterHub from it. This makes it easy to refer to the JupyterHub chart without having to use a long URL each time.

```
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 you can install the chart! Run this command from the directory that contains the `config.yaml` file to spin up JupyterHub:

```
helm install jupyterhub/jupyterhub \
    --version=v0.6 \
    --name=<YOUR-RELEASE-NAME> \
    --namespace=<YOUR-NAMESPACE> \
    -f config.yaml
```

where:

- `--name` is an identifier used by helm to refer to this deployment. You need it when you are changing the configuration of this install or deleting it. Use something descriptive that you will easily remember. For a class called *data8* you might wish set the name to **data8-jupyterhub**. In the future you can find out the name by using `helm list`.

- `--namespace` is an identifier used by Kubernetes (among other things) to identify a particular application that might be running on a single Kubernetes cluster. You can install many applications into the same Kubernetes cluster, and each instance of an application is usually separated by being in its own namespace. You'll need the namespace identifier for performing any commands with `kubectl`.

We recommend providing the same value to `--name` and `--namespace` for now to avoid too much confusion, but advanced users of Kubernetes and helm should feel free to use different values.

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 <YOUR-NAMESPACE>` and try again.

- In general, if something goes *wrong* with the install step, delete the Helm namespace 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` 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., v0.6 of the helm chart runs JupyterHub v0.8.1.

---

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

```
kubectl --namespace=<YOUR-NAMESPACE> get pod
```

4. Wait for the hub and proxy pod to begin running.

5. You can find the IP to use for accessing the JupyterHub with:

```
kubectl --namespace=<YOUR-NAMESPACE> get svc
```

The external IP for the `proxy-public` service should be accessible in a minute or two.

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

```
kubectl --namespace=<YOUR-NAMESPACE> describe svc proxy-public --output=wide
```

---

6. 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 JupyterHub running, you can extend it in many ways. You can use a pre-built image for the user container, build your own image, configure different authenticators, and more!

2.4 Turning Off JupyterHub and Computational Resources

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 resources on various cloud providers.

Tearing down your JupyterHub entails:

1. Deleting your Kubernetes namespace, which deletes all objects created and managed by Kubernetes

2. Deleting any computational 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)

2.4.1 For all cloud providers

Delete the helm namespace

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 to make your jupyterhub.

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

2. Next, delete the 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>
```

2.4.2 Google Cloud Platform

1. Perform the steps in *Delete the helm namespace*. These cloud provider agnostic steps will delete the helm chart and delete the hub's 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:

   (a) Compute -> Compute Engine -> Disks

   (b) Compute -> Kubernetes Engine -> Container Clusters

   (c) Tools -> Container Registry -> Images

   (d) 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.

2.4.3 Microsoft Azure AKS

1. Perform the steps in *Delete the helm namespace*. These cloud provider agnostic steps will delete the helm chart and delete the hub's 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.

2.4.4 Amazon Web Services (AWS)

1. Perform the steps in *Delete the helm namespace*. These cloud provider agnostic steps will delete the helm chart and delete the hub's namespace. This must be done before proceeding.

2. on CI host:

```
kops delete cluster <CLUSTER-NAME> --yes
exit #(leave CI host)
Terminicate CI Host
aws ec2 stop-instances --intance-ids <aws-instance id of CI HOST>
aws ec2 terminate-instances --instance-ids <aws-instance id of CI HOST>
```

Note: cluster name was set as an env var aka: NAME=<somename>.k8s.local Stopping the CI host will still incure disk storage and Ip address costs, but the host can be restarted at a later date to resume using.

Note: 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.

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 *Helm Chart Configuration Reference* for a list of frequently used configurable helm chart fields.

3.1 Extending your JupyterHub setup

The helm chart used to install JupyterHub has a lot of options for you to tweak. For a semi-complete list of the changes you can apply via your helm-chart, see the *Helm Chart Configuration Reference*.

3.1.1 Applying configuration changes

The general method to modify your Kubernetes deployment is to:

1. Make a change to the `config.yaml`

2. Run a helm upgrade:

```
helm upgrade <YOUR_RELEASE_NAME> jupyterhub/jupyterhub --version=v0.6 -f␣
↪config.yaml
```

   Where `<YOUR_RELEASE_NAME>` is the parameter you passed to `--name` when installing jupyterhub with `helm install`. If you don't remember it, you can probably find it by doing `helm list`.

3. Wait for the upgrade to finish, and make sure that when you do `kubectl --namespace=<YOUR_NAMESPACE> get pod` the hub and proxy pods are in `Ready` state. Your configuration change has been applied!

For information about the many things you can customize with changes to your helm chart, see *Customizing the User Environment*, *User Resources*, and *Helm Chart Configuration Reference*.

3.2 Customizing the User Environment

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

This page contains instructions for a few common ways you can extend the user experience for your kubernetes deployment.

The **user environment** is the set of 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 RStudio, RISE, JupyterLab, and others.

Usually a *docker image* specifies the functionality and environment that you wish to provide to users. The following sections will describe how to use existing Docker images, how to create custom images, and how to set environment variables.

3.2.1 Use an existing Docker image

Note: The Docker image you are using must have the `jupyterhub` package installed in order to work. Moreover, the version of `jupyterhub` must match the version installed by the helm chart that you're using. For example, `v0.6` of the helm chart uses `jupyterhub==0.8.1`.

Note: You can find the configuration for the default Docker image used in this guide here.

Using an existing Docker image, that someone else has written and maintained, is the simplest approach. For example, Project Jupyter maintains the jupyter/docker-stacks repo, which contains ready to use Docker images. Each image includes a set of commonly used science and data science libraries and tools.

The scipy-notebook image, which can be found in the `docker-stacks` repo, contains useful scientific programming libraries pre-installed. This image may satisfy your needs. If you wish to use an existing image, such as the `scipy-notebook` image, complete these steps:

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

```
singleuser:
  image:
    name: jupyter/scipy-notebook
    tag: c7fb6660d096
```

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

   Always use an explicit `tag`, such as a specific commit.

   Avoid using `latest`. Using `latest` 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. These directions will **pre-pull** the image to all the nodes in your cluster. This process may take several minutes to complete.

Note: Docker images must have the `jupyterhub` package installed within them to be used in this manner.

3.2.2 Build a custom Docker image with `repo2docker`

If you can't find a pre-existing image that suits your needs, you can create your own image. The easiest way to do this is with the package *repo2docker*.

Note: repo2docker lets you quickly convert a GitHub repository into a Docker image that can be used as a base for your JupyterHub instance. Anything inside the GitHub repository will exist in a user's environment when they join your JupyterHub:

- If you include a `requirements.txt` file in the root level of the repository, `repo2docker` will `pip install` the specified packages into the Docker image to be built.

- If you have an `environment.yaml` file, `conda` will create an environment based on this file's specification.

- If you have a `Dockerfile`, `repo2docker` will ignore everything else and just use the Dockerfile.

Below we'll cover how to use `repo2docker` to generate a Docker image and how to configure JupyterHub to build off of this image:

1. **Download and start Docker.** You can do this by [downloading and installing Docker](#). Once you've started Docker, it will show up as a tiny background application.

2. **Install repo2docker** using `pip`:

   ```
   pip install jupyter-repo2docker
   ```

   If that command fails due to insufficient permissions, try it with the command option, `user`:

   ```
   pip install --user jupyter-repo2docker
   ```

3. **Create (or find) a GitHub repository you want to use.** This repo should have all materials that you want your users to be able to use. You may want to include a [pip](#) `requirements.txt` file to list packages, one per file line, to install such as when using `pip install`. Specify the versions explicitly so the image is fully reproducible. An example `requirements.txt` follows:

   ```
   jupyterhub==0.8.*
   numpy==1.12.1
   scipy==0.19.0
   matplotlib==2.0
   ```

   As noted above, the requirements must include `jupyterhub`, pinned to a version compatible with the version of JupyterHub used by Helm chart.

4. **Use repo2docker to build a Docker image.**

   ```
   jupyter-repo2docker --user-name=jovyan --image=gcr.io/<PROJECT-NAME>/<IMAGE-NAME>:
   ↪<TAG> --no-run <YOUR-GITHUB-REPOSITORY>
   ```

   This tells `repo2docker` to fetch `master` of the GitHub repository, and uses heuristics to build a docker image of it.

   Note:

   - The project name should match your google cloud project's name.

   - Don't use underscores in your image name. Other than this, the name can be anything memorable. *This bug with underscores will be fixed soon.*

- The tag should be the first 6 characters of the SHA in the GitHub commit desired for building the image since this improves reproducibility.

5. **Push the newly-built Docker image to the cloud.** You can either push this to Docker Hub or to the gcloud docker repository. Here we'll demonstrate pushing to the gcloud repository:

```
gcloud docker -- push gcr.io/<project-name>/<image-name>:<tag>
```

6. **Edit the JupyterHub configuration to build from this image.** Edit `config.yaml` file to include these lines in it:

```
singleuser:
  image:
    name: gcr.io/<project-name>/<image-name>
    tag: <tag>
```

This step can be done automatically by setting a flag if desired.

7. **Tell helm to update JupyterHub to use this configuration.** Use the standard method to apply the changes to the config.

8. **Restart your notebook if you are already logged in.** If you already have a running JupyterHub session, you'll need to restart it (by stopping and starting your session from the control panel in the top right). New users won't have to do this.

Note: The contents of your GitHub repository might not show up if you have enabled persistent storage. Disable persistent storage if you want the GitHub repository contents to show up.

---

9. **Enjoy your new computing environment!** You should now have a live computing environment built off of the Docker image we've created.

3.2.3 Use JupyterLab by default

Warning: As JupyterLab is a quickly-evolving tool right now, it is important to use recent versions of JupyterLab. If you install JupyterLab with `conda`, **make sure to use the ''conda-forge'' channel instead of ''default''.**

JupyterLab is the next generation user interface for Project Jupyter. It can be used with JupyterHub, both as an optional interface and as a default.

In addition, a JupyterLab extension, called JupyterLab-Hub, provides a nice UI for accessing the JupyterHub control panel from JupyterLab. These instructions show how to install both JupyterLab and JupyterLab-Hub.

Note: If JupyterLab is installed on your hub (and with or without "JupyterLab Hub" installed), users can always switch to the classic Jupyter Notebook by selecting menu item "Help >> Launch Classic Notebook" or by replacing `/lab` with `/tree` in the URL (if the server is running). Similarly, you can access JupyterLab even if it is not the default by replacing `/tree` in the URL with `/lab`.

---

1. Install JupyterLab and the JupyterLab Hub extension in your user image, for example in your Dockerfile:

```
FROM jupyter/base-notebook:27ba57364579

...
ARG JUPYTERLAB_VERSION=0.31.12
RUN     pip install jupyterlab==$JUPYTERLAB_VERSION \
     && jupyter labextension install @jupyterlab/hub-extension
...
```

2. Enable JupyterLab in your Helm configuration by adding the following snippet:

```
hub:
  extraEnv:
    JUPYTER_ENABLE_LAB: 1
  extraConfig: |
    c.KubeSpawner.cmd = ['jupyter-labhub']
```

3. If you want users to launch automatically into JupyterLab instead of the classic notebook, set the following setting in your Helm configuration:

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

This will put users into JupyterLab when they launch their server.

Note: JupyterLab is in beta, so use with caution!

3.2.4 Set environment variables

Another way to affect your user's environment is by setting values for *environment variables*. While you can set them up in your Docker image, it is often easier to set them up in your helm chart.

To set them up in your helm chart, edit your `config.yaml` file 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 will read in an environment variable:

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

Other languages will have their own methods of reading these environment variables.

3.2.5 Pre-populating user's $HOME directory with files

When persistent storage is enabled (which is the default), the contents of the 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.

However, keep in mind that this command 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.

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

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 *Build a custom Docker image with repo2docker*)

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.

3.3 User Resources

Note: For a list of all the options you can configure with your helm chart, see the *Helm Chart 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.

3.3.1 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.

```
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!

3.3.2 Modifying user storage type and size

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

3.3.3 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.

3.4 User storage in JupyterHub

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.

`PersistentVolumeClaim`s and `PersistentVolume`s 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`.

3.4.1 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.

3.4.2 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 `PVC`s 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 `PersistentVolume`s 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 Disk](#)s. 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.

3.4.3 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!

3.5 User Management

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

3.5.1 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.

3.5.2 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
```
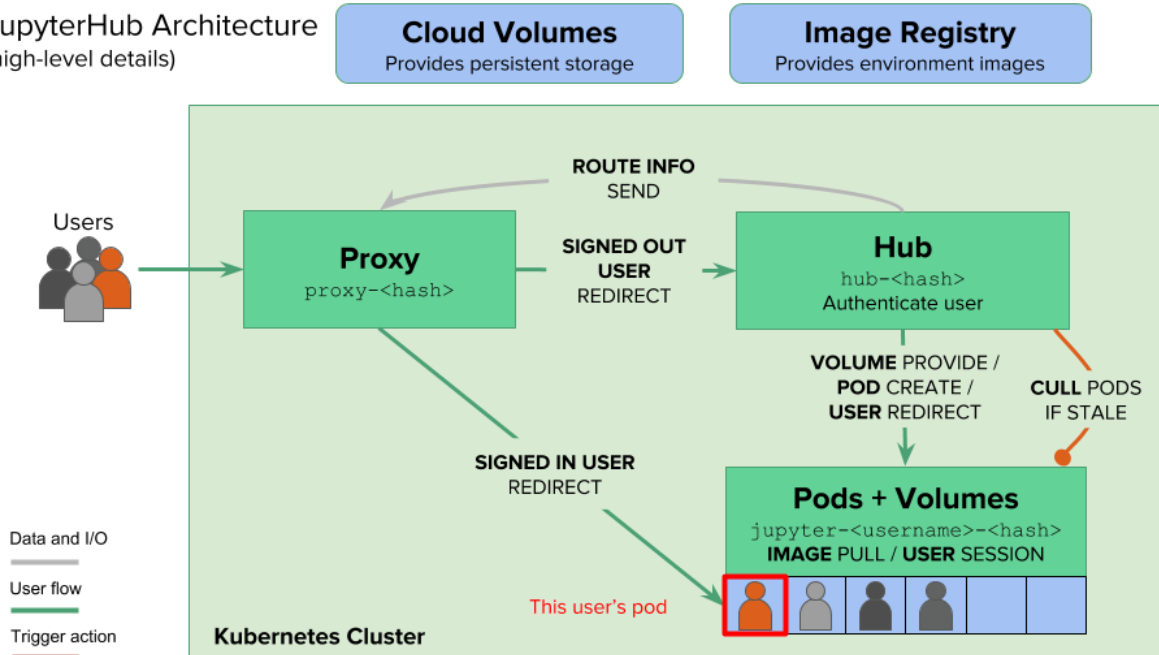
3.5.3 Authenticating Users

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

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.

4.1 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:

4.2 Debugging Kubernetes

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

4.2.1 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.

4.2.2 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
```

4.3 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.

4.3.1 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.

Below is the structure to use in order to authenticate with GitHub.

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

Giving access to organizations on GitHub

The configuration above will allow *any* GitHub user to access your JupyterHub. You can also restrict access to members of one or more GitHub organizations. To do so, see the configuration below.

```
auth:
  type: github
  github:
    ...
    org_whitelist:
      - "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
  github:
    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: |
    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"
```

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.

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
auth:
  type: custom
  custom:
    className: oauthenticator.generic.GenericOAuthenticator
    config:
      client_id: "y0urc1logonc1ient1d"
```

(continues on next page)

```
    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
```

4.3.2 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.

4.3.3 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'
        dnAttribute: 'cn'
      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'
    allowedGroups:
      - 'cn=researcher,ou=groups,dc=wikimedia,dc=org'
      - 'cn=operations,ou=groups,dc=wikimedia,dc=org'
```

4.3.4 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.

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

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

4.4 Speed and Optimization

This page contains information and guidelines for improving the speed, stability, and general optimization of your JupyterHub deployment.

4.4.1 Picking a Scheduler Strategy

Kubernetes offers very flexible ways to determine how it distributes pods on your nodes. The JupyterHub helm chart supports two common configurations, see below for a brief description of each.

Spread

- **Behavior**: This spreads user pods across **as many nodes as possible**.
- **Benefits**: A single node going down will not affect too many users. If you do not have explicit memory & cpu limits, this strategy also allows your users the most efficient use of RAM & CPU.
- **Drawbacks**: This strategy is less efficient when used with autoscaling.

This is the default strategy. To explicitly specify it, use the following in your `config.yaml`:

```
singleuser:
  schedulerStrategy: spread
```

Pack

- **Behavior**: This packs user pods into **as few nodes as possible**.
- **Benefits**: This reduces your resource utilization, which is useful in conjunction with autoscalers.
- **Drawbacks**: A single node going down might affect more user pods than using a "spread" strategy (depending on the node).

When you use this strategy, you should specify limits and guarantees for memory and cpu. This will make your users' experience more predictable.

To explicitly specify this strategy, use the following in your `config.yaml`:

```
singleuser:
  schedulerStrategy: pack
```

4.4.2 Pre-pulling

Pulling a user's images to a node forces a user to wait before the user's server is started. Sometimes, the wait can be 5 to 10 minutes. **Pre-pulling** the images on all the nodes can cut this wait time to a few seconds. Let's look at how pre-pulling works.

Hook - image pulling before upgrades

With the **pre-pulling hook**, which is enabled by default, the user's container image is pulled on all nodes whenever a `helm install` or `helm upgrade` is performed. While this causes `helm install` and `helm upgrade` to take several minutes as the update is scheduled after the pulling has completed, the users waiting time will decrease and become more reliable.

With the default helm upgrade settings, a `helm install` or `helm upgrade` will allow 5 minutes of image pulling before timing out. This wait time is configurable by passing the `--wait <seconds>` flag to the `helm` commands.

We recommend using pre-pulling. For the rare cases where you have a good reason to disable it, pre-pulling can be disabled. To disable the pre-pulling during `helm install` and `helm upgrade`, you can use the following snippet in your `config.yaml`:

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

Continuous - image pulling for added nodes

Cluster size can change through manual addition of nodes or autoscaling. When a new node is added to the cluster, the new node does not yet have the user image. A user using this new node would be forced to wait while the image is pulled from scratch. Ideally, it would be helpful to pre-pull images when the new node is added to the cluster.

With the **continuous pre-puller** enabled (disabled by default), the user's container image will be pre-pulled when a new node is added. New nodes can for example be added manually or by a cluster autoscaler. The **continuous pre-puller** uses a daemonset to force kubernetes to pull the user image on all nodes as soon as a node is present. The continuous pre-puller uses minimal resources on all nodes and greatly speeds up the user pod start time.

The continuous pre-puller is disabled by default. To enable it, use the following snippet in your `config.yaml`:

```
prePuller:
  continuous:
    enabled: true
```

Pre-pulling additional images

By default, the pre-puller only pulls the singleuser image & the networktools image (if access to cloud metadata is disabled). If you have customizations that need additional images present on all nodes, you can ask the pre-puller to also pull an arbitrary number of additional images.

```
prePuller:
  extraImages:
    ubuntu-xenial:
      name: ubuntu
      tag: 16.04
      policy: IfNotPresent
```

This snippet will pre-pull the `ubuntu:16.04` image on all nodes, for example. You can pre-pull any number of images.

4.5 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.

4.5.1 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.

4.5.2 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.

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!

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>
```
```

4.5.3 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.

4.5.4 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:

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

4.5.5 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
```

4.5.6 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`:

```
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 (particularly Azure AKS) do not support RBAC right now, and you might have to disable RBAC with this config to run on Azure.

4.5.7 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!

4.5.8 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:
    - 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.

4.6 Upgrading your JupyterHub Kubernetes deployment

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!

4.6.1 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.

4.6.2 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
```

4.6.3 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 <YOUR-HELM-RELEASE-NAME>.

4.7 FAQ

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

4.7.1 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.

4.7.2 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*.

4.8 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.

4.8.1 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!

4.8.2 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 += {
      "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
```

hub.extraConfigMap

This property takes a dictionary of values that are then made available for code in `hub.extraConfig` to read using a `z2jh.get_config` function. You can use this to easily separate your code (which goes in `hub.extraConfig`) from your config (which should go here).

For example, if you use the following snippet in your config.yaml file:

```
hub:
  extraConfigMap:
    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.

Note that the keys in `hub.extraConfigMap` must be alpha numeric strings starting with a character. Dashes and Underscores are not allowed.

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.

4.8.3 Picking a Scheduler Strategy

Kubernetes offers very flexible ways to determine how it distributes pods on your nodes. The JupyterHub helm chart supports two common configurations, see below for a brief description of each.

Spread

- **Behavior**: This spreads user pods across **as many nodes as possible**.

- **Benefits**: A single node going down will not affect too many users. If you do not have explicit memory & cpu limits, this strategy also allows your users the most efficient use of RAM & CPU.

- **Drawbacks**: This strategy is less efficient when used with autoscaling.

This is the default strategy. To explicitly specify it, use the following in your `config.yaml`:

```
singleuser:
    schedulerStrategy: spread
```

Pack

- **Behavior**: This packs user pods into **as few nodes as possible**.

- **Benefits**: This reduces your resource utilization, which is useful in conjunction with autoscalers.

- **Drawbacks**: A single node going down might affect more user pods than using a "spread" strategy (depending on the node).

When you use this strategy, you should specify limits and guarantees for memory and cpu. This will make your users' experience more predictable.

To explicitly specify this strategy, use the following in your `config.yaml`:

```
singleuser:
    schedulerStrategy: pack
```

4.8.4 Pre-pulling Images for Faster Startup

Pulling and building a user's images forces a user to wait before the user's server is started. Sometimes, the wait can be 5 to 10 minutes. **Pre-pulling** the images on all the nodes can cut this wait time to a few seconds. Let's look at how pre-pulling works.

Pre-pulling basics

With **pre-pulling**, which is enabled by default, the user's container image is pulled on all nodes whenever a `helm install` or `helm upgrade` is performed. While this causes `helm install` and `helm upgrade` to take several minutes, this time makes the user startup experience faster and more pleasant.

With the default **pre-pulling** setting, a `helm install` or `helm upgrade` will cause the system to wait for 5 minutes to begin pulling the images before timing out. This wait time is configurable by passing the `--wait <seconds>` flag to the `helm` commands.

We recommend using pre-pulling. For the rare cases where you have a good reason to disable it, pre-pulling can be disabled. To disable the pre-pulling during `helm install` and `helm upgrade`, you can use the following snippet in your `config.yaml`:

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

Pre-pulling and changes in cluster size

Cluster size can change through manual addition of nodes or autoscaling. When a new node is added to the cluster, the new node does not yet have the user image. A user using this new node would be forced to wait while the image is pulled from scratch. Ideally, it would be helpful to pre-pull images when the new node is added to the cluster.

By enabling the **continuous pre-puller** (default state is disabled), the user image will be pre-pulled when adding a new node. When enabled, the **continuous pre-puller** runs as a daemonset to force kubernetes to pull the user image on all nodes as soon as a node is present. The continuous pre-puller uses minimal resources on all nodes and greatly speeds up the user pod start time.

The continuous pre-puller is disabled by default. To enable it, use the following snippet in your `config.yaml`:

```
prePuller:
  continuous:
    enabled: true
```

Pre-pulling additional images

By default, the pre-puller only pulls the singleuser image & the networktools image (if access to cloud metadata is disabled). If you have customizations that need additional images present on all nodes, you can ask the pre-puller to also pull an arbitrary number of additional images.

```
prePuller:
   extraImages:
     ubuntu-xenial:
       name: ubuntu
       tag: 16.04
       policy: IfNotPresent
```

This snippet will pre-pull the `ubuntu:16.04` image on all nodes, for example. You can pre-pull any number of images.

4.9 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.

4.9.1 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

4.9.2 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.

4.9.3 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.

4.9.4 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.

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: Contibuting 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.

5.1 Community-authored documentation

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.

5.1.1 Links to blog posts

5.1.2 Links to community project repos

5.2 Zero to JupyterHub Gallery of Deployments

This is a community maintained list of organizations / people using the Zero to JupyterHub guide / 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

5.3 Tips and command 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!

5.3.1 `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.

5.3.2 `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.

5.3.3 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>
```

5.3.4 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.

5.3.5 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.

5.3.6 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")

Reference

6.1 Helm Chart Configuration Reference

The JupyterHub helm chart is configurable so that you can customize your JupyterHub setup however you'd like. You can extend user resources, build off of different Docker images, manage security and authentication, and more.

Below is a description of the fields that are exposed with the JupyterHub helm chart. For more guided information about some specific things you can do with modifications to the helm chart, see the extending jupyterhub and user environment pages.

6.1.1 proxy

proxy.secretToken

A 64-byte cryptographically secure randomly generated string used to secure communications between the hub and the configurable-http-proxy.

This must be generated with `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

6.1.2 auth

auth.state

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 ;) 32B 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.

This can be generated with `openssl rand -hex 32`.

If encryption is unavailable, auth_state cannot be persisted.

auth.state.enabled

Enable persisting auth_state (if available). See: http://jupyterhub.readthedocs.io/en/latest/api/auth.html

6.1.3 singleuser

Options for customizing the environment that is provided to the users after they log in.

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.memory

Set Memory limits & guarantees that are enforced for each user. See: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

singleuser.memory.guarantee

singleuser.memory.limit

singleuser.imagePullPolicy

Set the imagePullPolicy on the singleuser pods that are spun up by the hub.

See the kubernetes docs for more info on what the values mean.

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.guarantee

singleuser.cpu.limit

singleuser.imagePullSecret

Create a custom image pull secret used for spawned users.

This secret is created in the same namespace as your jupyterhub deployment and will be used to pull your single user image.

singleuser.imagePullSecret.username

Name of the user you want to use to connect to your private registry.

Examples:

- alexmorreale

- alex@pfc.com

singleuser.imagePullSecret.registry

Name of the private registry you want to create a credential set for.

Examples:

- private.jfrog.io

- alexmorreale.privatereg.net

singleuser.imagePullSecret.password

Password of the user you want to use to connect to your private registry.

Examples:

- plaintextpassword

- abc123SECRETzyx098

6.1.4 hub

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.

Examples:

- 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.

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.cookieSecret

A 64-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. Value set here will override the value in `jupyterhub_cookie_secret`.

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.

This must be generated with `openssl rand -hex 32`.

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

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 dynamically created using the default [dynamic provisioner]. 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.selector

Selectors to set for the PVC containing the sqlite database.

Useful when you are using a static PVC.

TODO: Link to pvc selector docs.

hub.db.pvc.annotations

Annotations to apply to the PVC containing the sqlite database.

TODO: Link to pvc annotations

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.imagePullPolicy

Set the imagePullPolicy on the hub pod.

See the kubernetes docs for more info on what the values mean.

hub.extraEnv

Extra environment variables that should be set for the hub pod.

A list of EnvVar objects.

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.labels

Extra labels to add to the hub pod.

See the kubernetes documentation to learn more about labels.

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 just write the raw Python to do it here.

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: |
    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.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.

6.2 Official JupyterHub and Project Jupyter Documentation

- 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.

6.3 Tools used in a JupyterHub Deployment

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.

6.3.1 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.

6.3.2 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.

6.3.3 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 compatmentalize 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 in to 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.

6.3.4 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.

6.3.5 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 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.

6.4 Glossary

A partial glossary of terms used in this guide. For more complete descriptions of the components in JupyterHub, see *Tools used in a JupyterHub Deployment*. 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.

culler   A separate process that stops the user pods of users who have not been active in a configured interval.

docker image   A docker image is similar to a recipe that Docker can use to build a working space which gives users the tools, libraries, and capabilities to be productive.

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`.

persistent storage   A filesystem attached to a user pod that allows the user to store notebooks and files that persist across multiple logins.

repo2docker   A tool which lets you quickly convert a GitHub repository into a Docker image that can be used as a base for your JupyterHub instance.

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, **73**
authenticator, **73**

C
culler, **73**

D
docker image, **73**

E
environment variables, **73**

P
persistent storage, **73**

R
repo2docker, **73**