
Zero to JupyterHub

Release 0.3.1

Chris Holdgraf

Mar 16, 2022

CONTENTS

1	Getting started with JupyterHub	3
1.1	Deployment Guide	3
1.2	Extending and Customizing JupyterHub	3
1.3	Dependencies for Deploying a JupyterHub Instance	3
2	Questions or Suggestions?	5
3	Creating a Kubernetes Cluster	7
3.1	Setting up Kubernetes on Google Cloud	7
3.2	Setting up Kubernetes on Microsoft Azure Container Service (ACS)	8
3.3	Next Step	9
4	Setting up Helm	11
4.1	Installation	11
4.2	Initialization	11
4.3	Next Step	11
5	Setting up JupyterHub	13
5.1	Prepare configuration file	13
5.2	Install JupyterHub	14
6	Turning Off JupyterHub and Computational Resources	15
7	Extending your JupyterHub setup	17
7.1	Applying configuration changes	17
7.2	Using an existing image	17
7.3	Setting memory and CPU guarantees / limits for your users	18
7.4	Extending your software stack with s2i	18
7.5	Pre-populating <i>\$HOME</i> directory with notebooks when using Persistent Volumes	20
7.6	Authenticating with OAuth2	20
7.7	Full Example of Google OAuth2	21
7.8	Expanding and contracting the size of your cluster	21
8	Tools used in a JupyterHub Deployment	23
8.1	Cloud Computing Providers	23
8.2	Container Technology	23
8.3	Kubernetes	24
8.4	Helm	26
8.5	JupyterHub	26
9	Resource management	29

10	Estimating costs	31
10.1	Computational Resources	31
10.2	Users	31
10.3	User usage patterns	31
10.4	Examples	32
11	Backups	33
12	Upgrading	35
13	Security Considerations	37
14	Troubleshooting	39
14.1	FAQ - General	39
14.2	Common error messages	39
14.3	Investigating Issues	40
15	Glossary	41
16	Additional resources	43

JupyterHub is a tool that allows you to quickly utilize cloud computing infrastructure to manage a hub that enables 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), 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 see anything that is incorrect or have any questions, feel free to reach out at the [issues page](#).

Creating your JupyterHub

GETTING STARTED WITH JUPYTERHUB

The goal of **JupyterHub** is to 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. 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.

1.1 Deployment Guide

We've put together a short walkthrough going from having nothing set up to a complete deployment of jupyterhub on Google Cloud. If you want to follow that comprehensive walkthrough, the next step on your journey is to [create a Kubernetes cluster on Google Cloud](#).

1.2 Extending and Customizing JupyterHub

If you'd like to know how to expand and customize your jupyterhub setup, such as increasing the computational resources available to users or changing authentication services, check out [Extending your JupyterHub setup](#).

1.3 Dependencies for Deploying a JupyterHub Instance

For a more extensive description of the tools and services that JupyterHub depends upon, see our [Tools used in a JupyterHub Deployment](#) page.

QUESTIONS OR SUGGESTIONS?

If you have questions or suggestions, please reach out at our [issues page](#) on GitHub.

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:

- *Google Cloud*
- *Microsoft Azure*
- Amazon EC2
- Red Hat OpenShift
- Others

3.1 Setting up Kubernetes on Google Cloud

[Google Container Engine](#) (confusingly abbreviated to 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. 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>.
2. Click the hamburger icon in the top left (the icon has three horizontal lines in one button). Go to “Billing” then “Payment Methods”, and make sure you have a credit card linked to the account. (You may also receive \$300 in trial credits.)
3. Install and initialize the **gcloud command-line tools**. These tools send commands to Google Cloud and lets you do things like create and delete clusters.
 - Go to the [gcloud downloads page](#) to **download and install the gcloud SDK**.
 - See the [gcloud documentation](#) for more information on the gcloud SDK.
 - Install `kubectl`, which is a tool for controlling kubernetes. From the terminal, enter:

```
gcloud components install kubectl
```

4. Create a Kubernetes cluster on Google Cloud, by typing in the following command:

```
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](#).

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

```
kubectl get node
```

The response should list three running nodes.

3.2 Setting up Kubernetes on Microsoft Azure Container Service (ACS)

Note: This is an alpha work-in-progress - please do not use in production! Help from people with more Azure experience would be highly welcome :)

1. Install and initialize the **Azure command-line tools**, which send commands to Azure and let you do things like create and delete clusters.
 - Go to the [azure-cli github repo](#) to download and install the **azure-cli** tools.
 - See the [az documentation](#) for more information on using the **az** tool with the Azure Container Service.
2. Authenticate the **az** tool so it may access your Azure account:

```
az login
```

3. Specify a [Azure resource group](#), and create one if it doesn't already exist:

```
export RESOURCE_GROUP=<YOUR_RESOURCE_GROUP>
export LOCATION=<YOUR_LOCATION>
az group create --name=${RESOURCE_GROUP} --location=${LOCATION}
```

where:

- `--name` specifies your Azure resource group. If a group doesn't exist, **az** will create it for you.
- `--location` specifies which computer center to use. To reduce latency, choose a zone closest to whoever is sending the commands. View available zones via `az account list-locations`.

5. Install **kubectl**, a tool for controlling Kubernetes:

```
az acs kubernetes install-cli
```

6. Create a Kubernetes cluster on Azure, by typing in the following commands:

```
export CLUSTER_NAME=<YOUR_CLUSTER_NAME>
export DNS_PREFIX=<YOUR_PREFIX>
az acs create --orchestrator-type=kubernetes \
  --resource-group=${RESOURCE_GROUP} \
```

(continues on next page)

(continued from previous page)

```
--name=${CLUSTER_NAME} \  
--dns-prefix=${DNS_PREFIX}
```

7. Authenticate kubectl:

```
az acs kubernetes get-credentials \  
  --resource-group=${RESOURCE_GROUP} \  
  --name=${CLUSTER_NAME}
```

where:

- `--resource-group` specifies your Azure resource group.
- `--name` is your ACS cluster name.
- `--dns-prefix` is the domain name prefix for the cluster.

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

```
kubectl get node
```

The response should list three running nodes.

3.3 Next Step

Now that you have a Kubernetes cluster running, it is time to *set up helm*.

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.

4.1 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.

4.2 Initialization

After installing helm on your machine, initialize helm on your Kubernetes cluster. At the terminal, enter:

```
helm init
```

This command only needs to run once per Kubernetes cluster.

4.3 Next Step

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

SETTING UP JUPYTERHUB

Now that we have a [Kubernetes cluster](#) and [helm](#) setup, we can begin setting up a JupyterHub.

5.1 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 two random hex strings to use as security tokens. Run these two commands (they're the same command but run them twice) in a terminal:

```
openssl rand -hex 32
openssl rand -hex 32
```

Copy the output each time, we'll use these hex strings 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 each occurrence of `RANDOM_STRING_N` below with the output of `openssl rand -hex 32`. The random hex strings are tokens that will be used to secure your JupyterHub instance (make sure that you keep the quotation marks):

```
hub:
  # output of first execution of 'openssl rand -hex 32'
  cookieSecret: "RANDOM_STRING_1"
token:
  # output of second execution of 'openssl rand -hex 32'
  proxy: "RANDOM_STRING_2"
```

For example:

```
hub:
  cookieSecret: "cb0b45df678709c5cc780ed73690898f7ba0659902f996017296143976ffb97c"
token:
  proxy: "712c4c6c0e78c6c745cfb126f5bbc4b9ba763c78b4bba5797e2eaf508ac99475"
```

4. Save the `config.yaml` file. If using the nano editor, hit `Ctrl-X` and make sure to answer 'yes' when it asks you to save.

5.2 Install JupyterHub

1. Let's use helm to create the instances that you configured with the `config.yaml` file. Run this command from the directory that contains the `config.yaml` file to spin up JupyterHub:

```
helm install https://github.com/jupyterhub/helm-chart/releases/download/v0.3.1/  
↪ jupyterhub-v0.3.1.tgz \\  
  --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: If you get a `release named <YOUR_CHART> already exists error`, then you should delete this helm-chart by running `helm delete --purge <YOUR_CHART>`. Then reinstall by repeating this step.

2. While Step 1 is running, you can see the pods being created by entering in a different terminal:

```
kubectl --namespace=<YOUR_NAMESPACE> get pod
```

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

5. 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!

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.

1. First, 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.

```
kubect1 delete namespace <your-namespace>
```

2. Next, you should 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 make sure everything has been deleted.

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

1. Compute Engine -> Disks
2. Container Engine
3. Networking -> 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.

Customization Guide

JupyterHub can be configured and customized to fit a variety of deployment requirements. This guide helps outline how to customize and extend a JupyterHub deployment.

EXTENDING YOUR JUPYTERHUB SETUP

The helm chart used to install JupyterHub has a lot of options for you to tweak. This page lists some of the most common changes.

7.1 Applying configuration changes

The general method is:

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

```
helm upgrade <YOUR_RELEASE_NAME> https://github.com/jupyterhub/helm-chart/  
↪releases/download/v0.3/jupyterhub-v0.3.tgz -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!

7.2 Using an existing image

It's possible to build your JupyterHub deployment off of a pre-existing Docker image. To do this, you need to find an existing image somewhere (such as DockerHub), and configure your installation to use it.

For example, UC Berkeley's [Data8 Program](#) publishes the image they are using on Dockerhub. To instruct JupyterHub to use this image, simply add this to your `config.yaml` file:

```
singleuser:  
  image:  
    name: berkeleydsep/singleuser-data8  
    tag: v0.1
```

You can then *apply the change* to the config as usual.

7.3 Setting memory and CPU guarantees / limits for your users

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).

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

7.4 Extending your software stack with s2i

s2i, also known as [Source to Image](#), lets you quickly convert a GitHub repository into a Docker image that we can use 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 your repository, s2i will `pip install` each of these packages into the Docker image to be built. Below we'll cover how to use s2i to generate a Docker image and how to configure JupyterHub to build off of this image.

Note: For this section, you'll need to install s2i and docker.

1. **Download s2i.** This is easily done with homebrew on a mac. For linux and Windows it entails a couple of quick commands that you can find in the links below:
 - On OSX: `brew install s2i`
 - On Linux and Windows: [follow these instructions](#)
2. **Download and start Docker.** You can do this by downloading and installing Docker at [this link](#). Once you've started Docker, it will show up as a tiny background application.
3. **Create (or find) a GitHub repository you want to use.** This repo should have all materials that you want your users to access. In addition you can include a `requirements.txt` file that has one package per line. These packages should be listed in the same way that you'd install them using `pip install`. You should also specify the versions explicitly so the image is fully reproducible. E.g.:

```
numpy==1.12.1
scipy==0.19.0
matplotlib==2.0
```

4. **Use `s2i` to build your Docker image.** `s2i` uses a template in order to know how to create the Docker image. We have provided one at the url in the commands below. Run this command:

```
s2i build --exclude "" <git-repo-url> jupyterhub/singleuser-builder-venv-3.5:v0.
↪1.5 gcr.io/<project-name>/<name-of-image>:<tag>
```

this effectively says `s2i, build`<this repository>` to a Docker image by using`<this template>` and call the image`<this>`. The --exclude "" ensures that all files are included in the container (e.g. .git directory).`

Note:

- The project name should match your google cloud project's name.
 - Don't use underscores in your image name. Other than this it can be anything memorable. This is a bug that will be fixed soon.
 - The tag should be the first 6 characters of the SHA in the GitHub commit for the image to build from.
-

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

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

6. **Edit the JupyterHub configuration to build from this image.** We do this by editing the `config.yaml` file that we originally created to include the jupyter hashes. Edit `config.yaml` by including these lines in it:

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

7. **Tell helm to update JupyterHub to use this configuration.** Using the normal method to *apply the change* to the config.
8. **Restart your notebook if you are already logging 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.
9. **Enjoy your new computing environment!** You should now have a live computing environment built off of the Docker image we've created.

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

7.5 Pre-populating `$HOME` directory with notebooks when using Persistent Volumes

By default, Persistent Volumes are used, so if you would like to include the contents of the GitHub repository in the `$HOME` directory (e.g. all of the `*.ipynb` files), then edit `config.yaml` include these lines in it:

```
singleuser:
  lifecycleHooks:
    postStart:
      exec:
        command: ["/bin/sh", "-c", "test -f $HOME/.copied || cp -Rf /srv/app/
↪src/. $HOME/; touch $HOME/.copied"]
```

Note that this will only copy the contents of the directory to `$HOME` *once* - the first time the user logs in. Further updates will not be reflected. There is work in progress for making this better.

7.6 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 two common authentication services. Note that in each case, you need to get the authentication credential information before you can configure the helmchart for authentication.

Google

For more information see the full example of Google OAuth2 in the next section.

```
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"
```

GitHub

```
auth:
  type: github
  github:
    clientId: "y0urg1thubclient1d"
    clientSecret: "an0therlongs3cretstring"
    callbackUrl: "http://<your_jupyterhub_host>/hub/oauth_callback"
```


7.7 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.

7.8 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. Use the `resize` command and provide a new cluster size 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`.

Note: When 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.
-

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.

8.1 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](#).

8.2 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:

8.2.1 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.

8.2.2 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.

8.3 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.

8.3.1 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.

8.3.2 Pods

Pods are essentially a collection of one or more *containers* that run together. You can think of them as a way of combining containers that, as a group, accomplish some goal.

For example, say you want to create a web server that is open to the world, but you also want authentication so that only a select group of users can access it. You could use a single pod with two containers.

- One that does the authentication. It would have something like Apache specified in its container image, and would be connected to the outside world.
- One that receives information from the authentication container, and does something fancy with it (maybe it runs a python process).

This is useful because it lets you compartmentalize the components of the service that you want to run, which makes things easier to manage and keeps things more stable.

For more information about pods, see the [Kubernetes documentation about pods](#).

8.3.3 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".

8.3.4 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](#).

8.3.5 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](#).

8.3.6 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](#).

8.4 Helm

Helm is a way of specifying kubernetes objects with a standard template.

8.4.1 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 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`.

8.4.2 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.

8.5 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 Machiner
- A bunch of user pods that are constantly being created or destroyed.

Below we’ll describe the primary JupyterHub pods.

8.5.1 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.

8.5.2 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.

Administrator Guide

This section provides information on managing and maintaining a staging or production deployment of JupyterHub.

RESOURCE MANAGEMENT

Under development

ESTIMATING 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

10.1 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. As a general rule, costs associated with RAM scale at <XXX> cost.

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. Storage roughly scales at <XXX> cost.

10.2 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.

10.3 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.

10.4 Examples

Here are a few examples that describe the use cases and amount of resources used by a particular JupyterHub implementation, and how much it might cost. Your estimates will vary.

10.4.1 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.

BACKUPS

Under development

UPGRADING

Under development

SECURITY CONSIDERATIONS

Under development

TROUBLESHOOTING

14.1 FAQ - General

I thought I had deleted my cloud resources, but they still show up. Why?

You probably deleted the specific nodes, but not the kubernetes cluster that was controlling those nodes. Kubernetes is designed to make sure that a specific set of resources is available at all times. This means that if you only delete the nodes, but not the kubernetes instance, then it will detect the loss of computers and will create two new nodes to compensate.

How does billing for this work?

JupyterHub isn't handling any of the billing for your usage. That's done through whatever cloud service you're using.

14.2 Common error messages

14.2.1 General

This section includes “provider agnostic” error messages for JupyterHub and Kubernetes.

14.2.2 Google Cloud

1. Could not find default credentials. See <https://developers.google.com/accounts/docs/application-default-credentials> for more information.

Execute `gcloud auth application-default login` and follow the prompts. The provided link in the error message has additional options for advanced use cases.

2. ERROR: (gcloud.container.clusters.create) ResponseError: code=503, message=Project staeiou-5f880 is not fully initialized with the default service accounts. Please try again later.

Go to <https://console.cloud.google.com/kubernetes/list> and click ‘enable’ and follow the prompts.

14.3 Investigating Issues

If you encounter any issues or wish to see what's happening under the hood, use the following commands.

To see running pods:

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

To see the logs:

```
kubectl --namespace=<YOUR-NAMESPACE> logs <pod-name>
```

You can pass `-f` option to the logs command to **tail** them.

Tip: Google Cloud: You can see the logs in the GUI on <https://console.cloud.google.com> there should be **logging** under the hamburger menu.

Reference

GLOSSARY

A partial glossary of terms used in this guide. For more complete descriptions of the components in JupyterHub, see [the list of tools used in JupyterHub](#). 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.

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

ADDITIONAL RESOURCES

Under development