**Dask: A Quick Introduction**

Dask is a flexible library for parallel computing in Python.

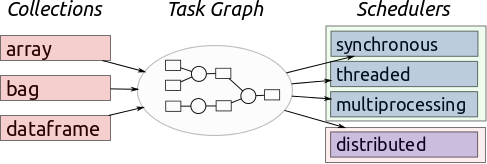
Dask is composed of two parts:

* Dynamic task scheduling optimized for computation. This is like Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
* “Big Data” collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

At its core Dask looks a lot like a tweaked Airflow/Luigi/Celery.  There is a central task scheduler that sends jobs (Python functions) out to lots of worker processes either on the same machine or on a cluster:

* Worker A, please compute x = f(1), Worker B please compute y = g(2).
* Worker A, when g(2) is done please get y from Worker B and compute z = h(x, y).

Where Dask differs is that while Airflow/Luigi/Celery were primarily designed for long-ish running data engineering jobs, Dask was designed for computation and interactive data science.  It is focused less on running jobs regularly (like Airflow) or integrating with lots of Data engineering services (like Luigi) and much more on millisecond response times, inter-worker data sharing, Jupyter notebook integration, etc. On top of this task scheduler, Dask has inbuilt higher-level array, dataframe, and list interfaces that mimic NumPy, Pandas, and Python iterators.  This pushes Dask into the space of the popular "Big Data" frameworks like Hadoop and Spark[[1]](#footnote-1).



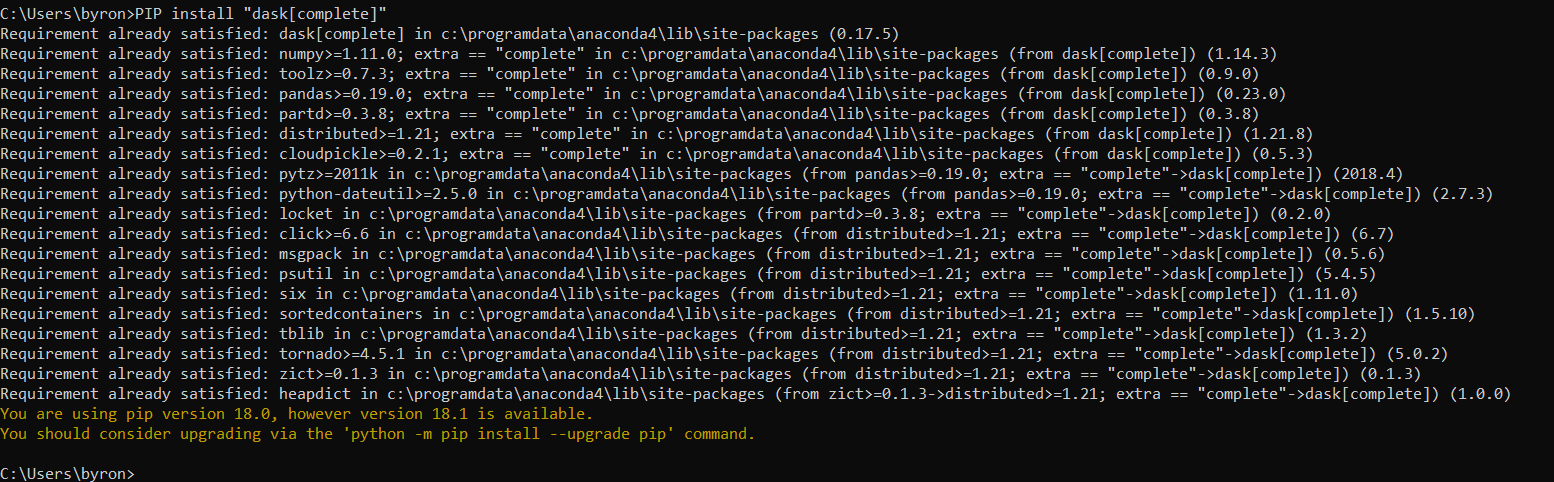
**Installing Dask**

**Local computer**:

You can install dask with conda, with pip, or by installing from source, using the following commands;

* conda install dask
* pip install "dask[complete]"

The installation with pip installs both Dask and dependencies like NumPy, Pandas, and so on that are necessary for different workloads.



**Installing Dask on the Cloud**

Installing dask on the cloud especially if you have a huge workload and would like to use a cluster to process the data can get a bit tricky. The installation process that we went through was through AWS and it required us to create a dns domain, create a cluster using kubernetes and then install dask on top of Kubernetes cluster.

***Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation[[2]](#footnote-2).***

We setup a Kubernetes cluster on AWS cloud environment using a tool called KOPS. kops is a provisioning system with; Fully automated installation, uses DNS to identify clusters, has self-healing: everything runs in Auto-Scaling Groups and has High-Availability support

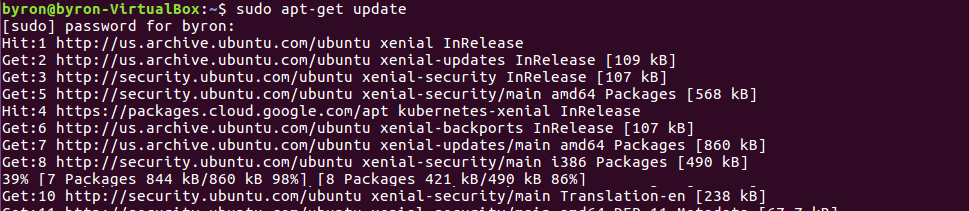
**2.1 Pre-requisites**

As a part of the setup, we needed an Ubuntu or Debian instance with latest updates and other supporting utility/tools like AWS-CLI, S3 bucket, Hosted Zone on Route 53 and a registered domain.

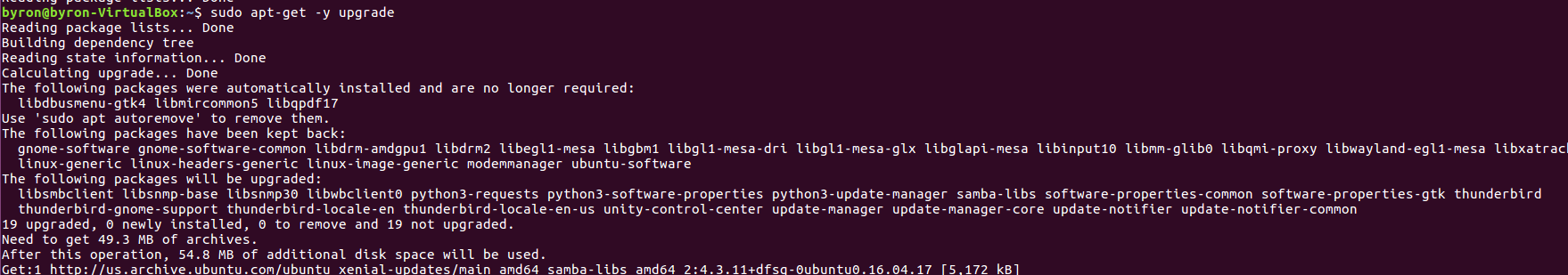
**Ubuntu Installation**

We installed an instance of Ubuntu using VirtualBox and made sure that it had the latest updates by running the following commands

**sudo apt-get update**: To update all the latest packages



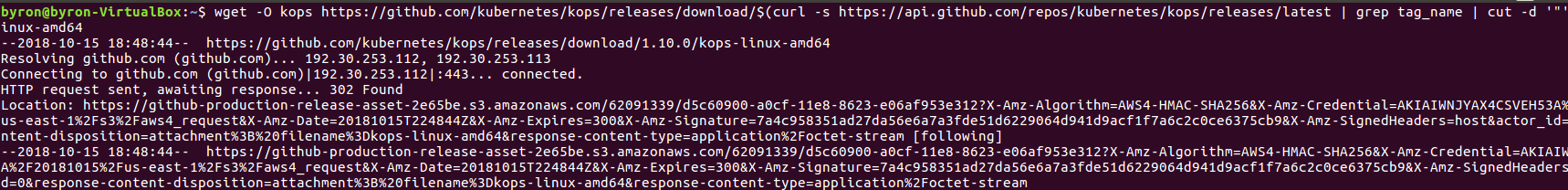
**sudo apt-get -y upgrade**: To ensure that all the present packages are upgraded



**Kops Installation**

We installed kops by running the command below changed the permission before moving it to /usr/local/bin;

1. **curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s** [**https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl**](https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl)
2. **chmod +x kubectl**
3. **sudo mv kubectl /usr/local/bin/**

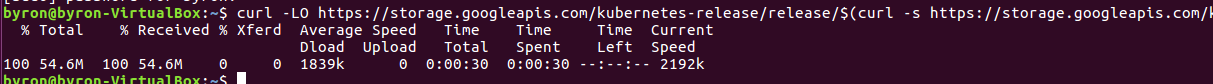


#### **kubectl installation**

We downloaded the latest version of the kubectl and changed the permission before moving it to /usr/local/bin

Kubectl installation commands

1. **curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl**
2. **chmod +x kubectl**
3. **sudo mv kubectl /usr/local/bin/**



#### **AWS CLI**

To install AWS CLI, run the following commands

1. **sudo apt-get install python-pip**
2. **pip install --upgrade pip**
3. **sudo pip install awscli**

#### **IAM user creation**

On AWS console, I have created a new IAM user (for example kops) with full access and save the access keys as it would be used to configure the AWS CLI. This role will be used to give your CI host permission to create and destroy resources on AWS

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

Create the IAM User using the following commands

You can create the kops IAM user from the command line using the following:

aws iam create-group --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/AmazonEC2FullAccess --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/AmazonRoute53FullAccess --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/IAMFullAccess --group-name kops

aws iam attach-group-policy --policy-arn arn:aws:iam::aws:policy/AmazonVPCFullAccess --group-name kops

aws iam create-user --user-name kops

aws iam add-user-to-group --user-name kops --group-name kops

aws iam create-access-key --user-name kops

Then we copied both AWS access key ID and AWS secret access key for later usage.

On the EC2 instance, I have configured the newly created AWS IAM user with following commands

**$aws configure**

AWS Access Key ID [None]:

AWS Secret Access Key [None]: Default region name [None]: < Optional : Please enter the region or blank for default>

Default output format [None]: < Optional : Please enter the output format or blank for default>

Run the ssh-keygen command to generate an ssh key. It will be used to connect to the kubernetes cluster which you are going to create. In my case, the ssh keys by default are stored in .ssh folder of the user home directory.

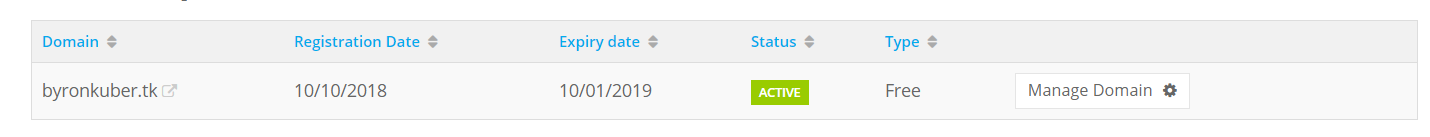
**ssh-keygen**

#### **Domain Creation**

We created a domain for the cluster user "kops" and DNS for discovery which will be used inside the cluster and to reach the kubernetes API server from the client. It should have a valid DNS name. An existing domain or a new domain can be used. In this example, I have created a new domain and hosted it on dot.tk which is a free domain provider.

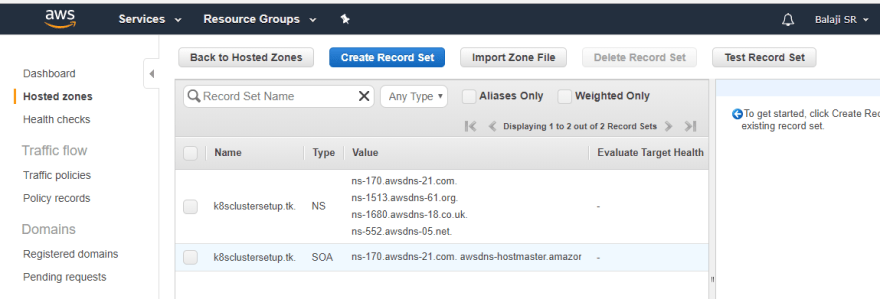
##### Domain registration on dot.tk / freenom.com:

We have registered a new domain on freenom.com with the name byronkuber.tk which I going to use for my Kubernetes cluster.

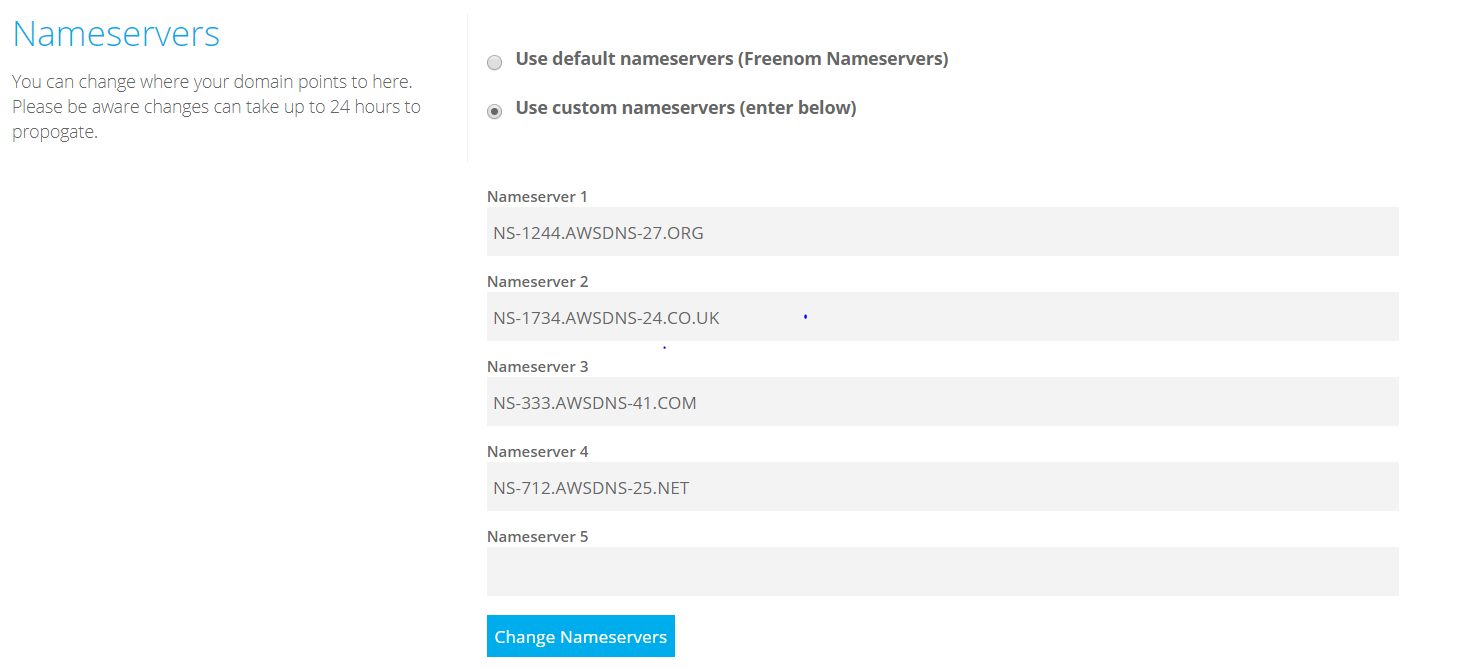


#### **Create Hosted Zone**

On AWS console, we created a new Hosted zone on router 53. I have logged into AWS console, navigate to router53 DNS management and created new Hosted Zone. It's advisable to create a subdomain. This creates a set of name servers which can be copied for later usage. I have copied the name server details which starts with ns-xxx.awsdns-xx.com, ns-xxx.awsdns-xx.co.uk, ns-xxx.awsdns-xx.org, ns-xxx.awsdns-xx.net.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--INZLU-Db--/c_limit,f_auto,fl_progressive,q_auto,w_880/https:/thepracticaldev.s3.amazonaws.com/i/u25buxcbudxxqzd83x8l.png)

These NameServer (NS) values should be updated on the domain service provider. In this setup, we have updated the NS details on freenom.com.



# **S3 bucket creation**

We created a new S3 bucket with a meaningful name (for example "kopsclusterdemo") which is used to store the cluster state. Kubernetes uses S3 to store the cluster details like configuration, keys, etc.

Create a bucket using the following command;

*bucket\_name=dodge-state-store aws s3api create-bucket --bucket ${bucket\_name}*

*--region us-east-1*

Here we created a bucket called dodge-state-store

With this, all the pre-requisite has been setup and the environment is ready to create and launch the Kubernetes cluster.

# **Install Helm**

Run the following command to install helm

$ sudo snap install helm

## **INSTALLING** **TILLER**

Tiller, the server portion of Helm, typically runs inside of your Kubernetes cluster. But for development, it can also be run locally, and configured to talk to a remote Kubernetes cluster.

### Easy In-Cluster Installation[[3]](#footnote-3)

The easiest way to install tiller into the cluster is simply to run helm init. This will validate that helm’s local environment is set up correctly (and set it up if necessary). Then it will connect to whatever cluster kubectl connects to by default (kubectl config view). Once it connects, it will install tiller into the kube-system namespace.

After helm init, you should be able to run kubectl get pods --namespace kube-systemand see Tiller running.

Create a cluster by running the following command

kops --v=3 create cluster [caya.byronkuber.tk](http://caya.byronkuber.tk/) --cloud=aws --state=s3://dodge-state-store --zones=us-west-1a --kubernetes-version=1.4.0 --node-count=3 --master-zones=us-west-1a --node-size=t2.small --master-size=t2.large –ssh-public-key=~/.ssh/id\_rsa.pub

here, we created a cluster called [caya.byronkuber.tk](http://caya.byronkuber.tk/)

# **Install Dask**

Next, we’re going to install a Dask chart. [Kubernetes Charts](https://github.com/kubernetes/charts) are curated application definitions for Helm. To install the Dask chart, we’ll update the known Charts channels before installing the stable version of Dask.

We need to create a serviceaccount  with API permissions. More details about how to do this are available in the [Kubernetes RBAC docs](https://kubernetes.io/docs/reference/access-authn-authz/rbac/#service-account-permissions).

*kubectl create serviceaccount --namespace kube-system tiller kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --serviceaccount=kube-system:tiller kubectl patch deploy --namespace kube-system tiller-deploy -p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}' helm init --service-account tiller –upgrade*

**Install Dask by running the following command**

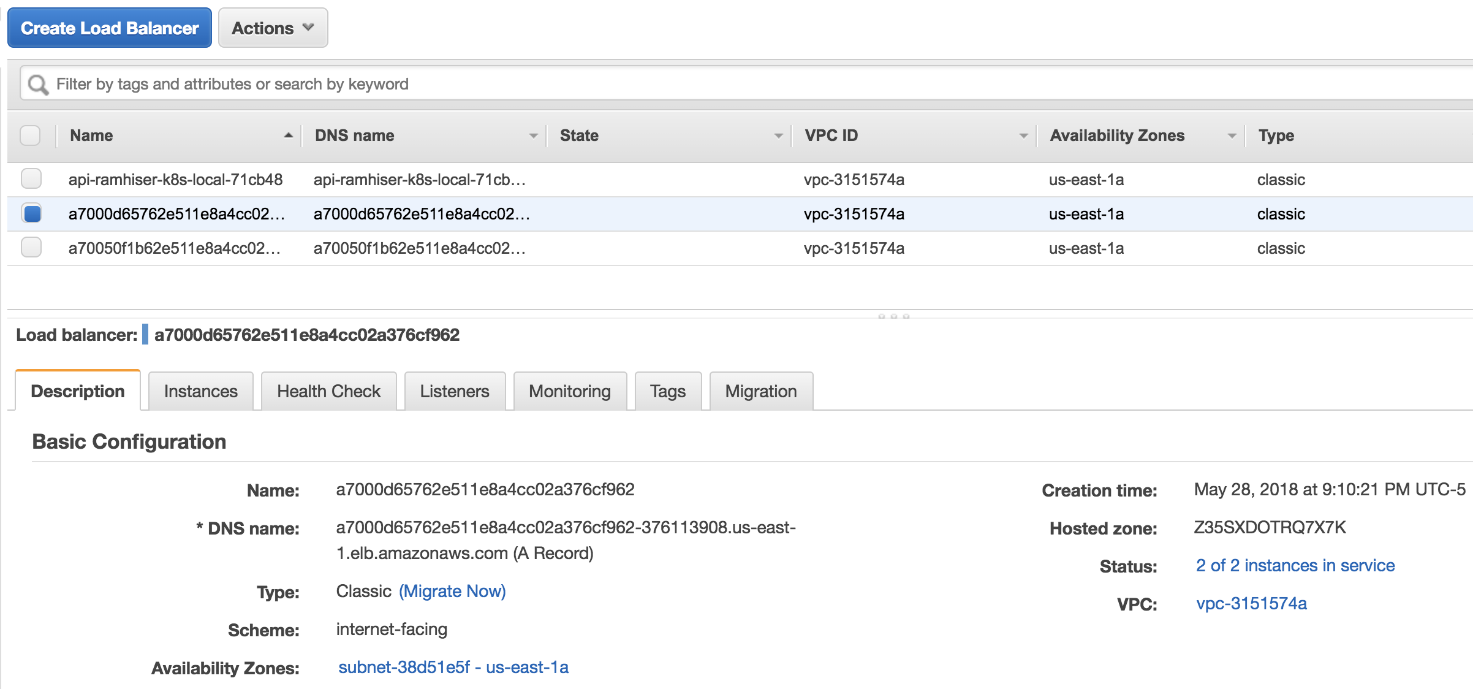
*helm install stable/dask*

Dask is now installed on the Kubernetes cluster. Notice that Helm has given our deployment the name xxxxxxx-xxx. The resources (pods and services) are all prepended with xxxxxxx-xxx as well.

As you can see, we launched a dask-scheduler, a dask-jupyter, and 3 dask-workerprocesses (default config). Below, we’ll walk through customizing the process.

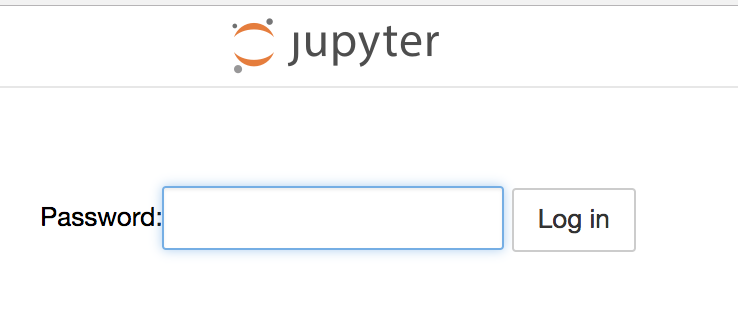
Also, notice the default Jupyter password: dask. We’ll use it to login to our Jupyter server later.

Notice that the EXTERNAL-IP displays hex values. These refer to [AWS ELB (Elastic Load Balancer)](https://aws.amazon.com/elasticloadbalancing/)entries you can find in your AWS console: EC2 -> Load Balancers. You can get the exposed DNS entry by matching the EXTERNAL-IP to the appropriate load balancer. For instance, the screenshot below shows that the DNS entry for the Jupyter node is http://xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.us-east-1.elb.amazonaws.com/.

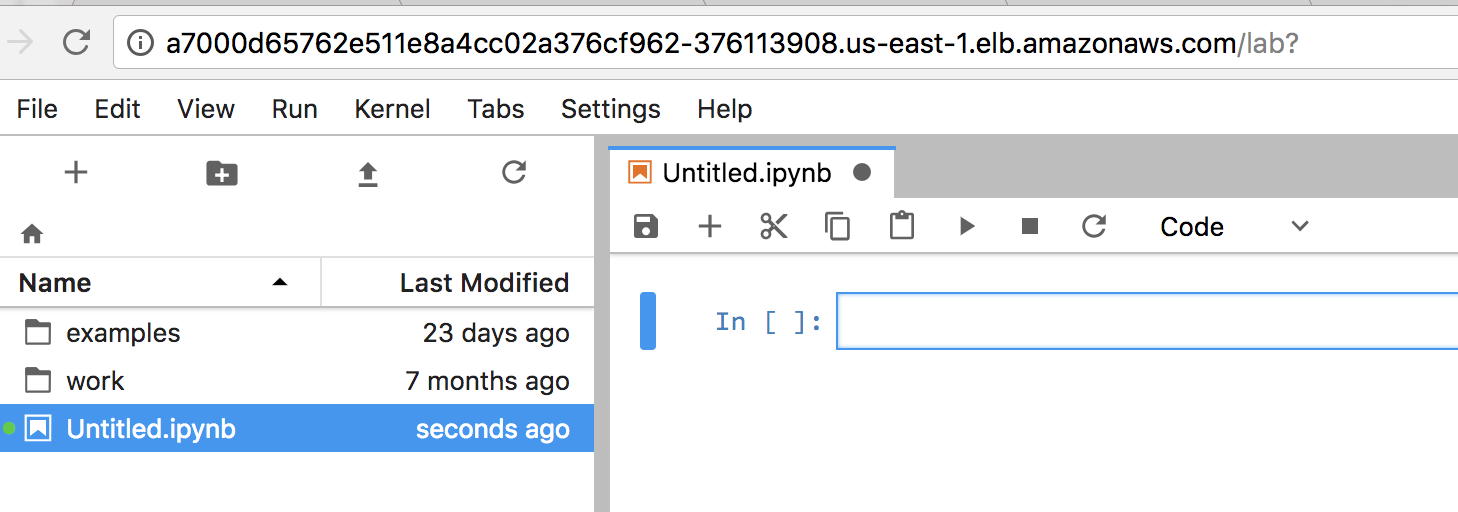


# Jupyter Server

Now that we have the DNS entry, let’s go to the Jupyter server in the browser at: http://xxxxxxxxxxxxxxxxxxxxxxxxxxxx.us-east-1.elb.amazonaws.com/. The first thing you’ll see is a Jupyter password prompt. Recall the default password is: dask.



After entering the password, you’ll see a running instance of [JupyterLab](https://github.com/jupyterlab/jupyterlab).



The notebooks include lots of useful information, such as:

* Parallelizing Python code with Dask
* Using [Dask futures](http://dask.pydata.org/en/latest/futures.html)
* Parallelizing Pandas operations with [Dask dataframes](http://dask.pydata.org/en/latest/dataframe.html)

Run a demo

Run the following commands to check the health and status of the cluster and to get the url to the Jupyter lab and the task scheduler

1. For the DNS status: Run dig ns [name](http://byronkuber.tk/) of ***dns created***
2. To view all the running nodes: *kubectl get nodes*
3. To check the ages of all nodes: *kubectl get pods*
4. To get information about jupyter and the web access: *kubectl get all*

**References**

<https://ramhiser.com/post/2018-05-28-adding-dask-and-jupyter-to-kubernetes-cluster/>

<http://docs.dask.org/en/latest/_downloads/daskcheatsheet.pdf>

https://dev.to/sr\_balaji/kubernetes-cluster-setup-on-aws-13m6

1. https://www.kdnuggets.com/2016/09/introducing-dask-parallel-programming.html [↑](#footnote-ref-1)
2. https://blog.sourcerer.io/a-kubernetes-quick-start-for-people-who-know-just-enough-about-docker-to-get-by-71c5933b4633 [↑](#footnote-ref-2)
3. https://docs.helm.sh/using\_helm/#quickstart [↑](#footnote-ref-3)