CyVerse Documentation

Release 0.1.0

CyVerse

Sep 18, 2020

Container Camp Workshop

CHAPTER 1

Workshop Code of Conduct

All attendees, speakers, sponsors and volunteers at our workshop are required to follow this code of conduct. Organisers will enforce this code throughout the event. We expect cooperation from all participants to help ensure a safe environment for everyone.

This Code of Conduct is taken from http://confcodeofconduct.com/. See http://www.ashedryden.com/blog/codes-of-conduct-101-faq for more information on codes of conduct.

FAIR principles

Container Camp supports FAIR data principles by providing services that help make data Findable, Accessible, Interoperable, and Reusable. Participants will get an introduction to containers and learn how to create and manage containers.

Learning objectives

Participants will learn key containerization concepts for developing reproducible analysis pipelines, with emphasis on container lifecycle management from design to execution and scaling.

The workshop will cover key concepts about containers such as defining the architecture of containers, building images and pushing them to public and private repositories as well as how to scale your analysis from laptop to cloud and to HPC systems using containers.

Who should attend?

Faculty, researchers, postdocs, and graduate students who use and analyze data of all types (genomics, image data, from animals, plants, etc.).

Workshop level

This workshop is focused on beginner-level users with little to no previous container experience.

Intermeidate and advanced users who attend will gain a better understanding of and ability with container capabilities and resources, including deploying their own tools and extending these analyses into Cloud and HPC.

Need help?

Couldn't find what you were looking for?

- You can reach the lead instructor, Upendra Devisetty, at upendra at cyverse dot org.

- You can also talk to any of the instructors or TAs if you need immediate help.
- Chat with us and our community on Slack.
- Post an issue on the documentation issue tracker on GitHub

CHAPTER 2

Pre-Workshop Setup

Please complete the minimum Setup Instructions to prepare for the Container Camp workshop at CyVerse, The University of Arizona, which will run from March 7th to 9th, 2018.

Prerequisite	Notes	Links
Wi-Fi-enabled laptop	You should be able to use any laptop using Windows/MacOS/Linux. We **strongly recommend** Firefox or Chrome browser; **We do not recommend Microsoft Edge Browser**. It is recommended that you have administrative/install permissions on your laptop.	• Download FireFox • Download Chrome
CyVerse Account	Please ensure that you have a CyVerse account and have **verfied** your account by completing the verification steps in the email you got when you registered.	Register for your cyverse account at http://user.cyverse.org/. You can **test your account** by logging into http://user.cyverse.org/.
Github Account	Please ensure that you have a Github account if you don't have one already	Register for your Github account at https://github.com/.
Dockerhub Account	Please ensure that you have a Dockerhub account if you don't have one already	Register for your Dockerhub account at https://hub.docker.com/.
Quay.io Account	This is completely optional but recommended and can use github to log-in to quay.io account	Register for your Quay.io account at https://quay.io/.
XSEDE Account	Please ensure that you have a XSEDE account by registering at XSEDE portal to access Jetstream cloud computing	Register for your XSEDE account at https://portal.xsede.org/.
TACC Account	Please ensure that you have a TACC account to access Stampede2 HPC computer at TACC	Register for your TACC account at https://portal.tacc.utexas.edu/.
Text Editor	Please ensure that you have a Text editor of your choice. Any decent text editor would be sufficient and recommended ones include Sublime2 and Atom	• Register for Sublime at https://www.sublimetext.com/. • Register for Atom at https://atom.io/.
Slack for networking	We will be using Slack extensively for communication and networking purposes	Register for Slack at https://slack.com/.

Optional Download Extras

Listed below are some extra downloads that aren't required for the workshop, but which provide some options for functionalities we will cover.

Tool	Notes	Link
SSH Clients (Windows)	PuTTY allows SSH connection to a remote machine, and is designed for Windows users who do not have a Mac/Linux terminal. MobaXterm is a single Windows application that provides a ton of functions for programmers, webmasters, IT administrators, and anybody is looking to manage system remotely	• Download PuTTY • Download mobaXterm • Update Windows 10 to use Linux Bash
Cyberduck	Cyberduck is a third-party tool for uploading/downloading data to CyVerse Data Store. Currently, this tool is available for Windows/MacOS only. You will need to download Cyberduck and the connection profile. We will go through configuration and installation at the workshop.	• Download Cyberduck • Download CyVerse Cyberduck connection profile
iCommands	iCommands are third-party software for command-line connection to the CyVerse Data Store.	Download and installation instructions available at CyVerse Learning Center

CHAPTER 3

Agenda

Below are the schedule and classroom materials for Container Camp at The University of Arizona, which will run from March 7th to 9th, 2018. The workshop will take place in Drachman A116. Click this link to the building - https://goo.gl/7Yv4PA

This workshop runs under a Code of Conduct. Please respect it and be excellent to each other!

Twitter hash tag: #cc2018

We will use this for notetaking - https://goo.gl/6Bd9tX

Day	Time	Topic/Activity	Notes/Links
03/07/18 (Wednesday)	8:30-9:00	General introduction to CyVerse and Camp logistics (Nirav Merchant & Upendra Devisetty)	Intro slides
	9:00-9:30	General overview of container technology landscape (Nirav Merchant)	Intro slides
	9:30-9:45	Coffee and snack break with networking	served in A127-29 across the hall (pls no food/bev in A116)
	9:45-11:00	Introduction to Docker (Kapeel Chougule)	• Docker intro slides • Docker intro demo
	11:00-12.00	Singularity (Vanessa Sochat)	Remote talk
	12:00-1:00	Lunch break on your own	
	1:00-2:30	Advanced Docker (Upendra Devisetty)	Advanced docker

Continued on next page

Table 1 – continued from previous page

Day	Time	Topic/Activity	Notes/Links
	2:30-3:00	Coffee break (15 min) and afternoon session planning	served in A127-29
	3:00-5:00	BYOD (Hands on Project)	
	5:00-6:00	Catchup with instructors	
03/08/18 (Thursday)	8:30-9:00	Recap and planning day 2	
	9:00-9:30	General overview of Singularity (John Fonner)	Singularity Overview Slides
	9:30-10.00	Singularity setup (Tyson Swetnam)	Singularity Introduction
	10:00-10:30	Coffee + snack break with networking	served in A127-29
	10:30-12.00	Singularity basics (Tyson Swetnam)	• Gitpitch slides • University of Arizona High Performance Computing • Introduction to Singularity
	12:00-1:00	Lunch break on your own	
	1:00-2:30	Advanced Singularity (John Fonner)	Advanced Singularity
	2:30-3:00	Coffee + snack break with networking	served in A127-29
	3:00-5:00	BYOD (Hands on Project)	
	5:00-6:00	Catchup with instructors	
03/09/18 (Friday)	8:30-9:30	Day 2 review and putting it all together	
	9:30-10:00	500,000 containers a day? OSG Singularity Infrastructure (Mats Rynge)	• Slides • Exercises
	10.00-10:30	Pegasus Workflows with Application Containers (Mats Rynge)	• Slides • Exercises
	10:30-10:45	Coffee + snack break with networking	served in A127-29
	10:45-11:30	Distributed computing with containers (Nick Hazekamp & Kyle Sweeney **remotely**)	Introduction to Container scaling
	11:30-12:30	Lunch break	CyVerse-hosted Food Truck - stay tuned for instructions
	12:30-1:15	Distributed computing with containers (Nick Hazekamp & Kyle Sweeney **remotely**)	Introduction to Container scaling

Continued on next page

Table 1 – continued from previous page

Day	Time	Topic/Activity	Notes/Links
	1:15-2:30	Biocontainers (Upendra Devisetty & John Fonner)	• Bicontainers Slides • Biocontainers Hands-on • Biocontainers on HPC
	2:30-3:00	Coffee + snack break with networking	served in A127-29
	3:00-5:00	BYOD (Hands on Project) and end of workshop	

CHAPTER 4

About CyVerse

CyVerse Vision: Transforming science through data-driven discovery.

CyVerse Mission: Design, deploy, and expand a national cyberinfrastructure for life sciences research and train scientists in its use. CyVerse provides life scientists with powerful computational infrastructure to handle huge datasets and complex analyses, thus enabling data-driven discovery. Our powerful extensible platforms provide data storage, bioinformatics tools, image analyses, cloud services, APIs, and more.

While originally created with the name iPlant Collaborative to serve U.S. plant science communities, CyVerse cyberinfrastructure is germane to all life sciences disciplines and works equally well on data from plants, animals, or microbes. By democratizing access to supercomputing capabilities, we provide a crucial resource to enable scientists to find solutions for the future. CyVerse is of, by, and for the community, and community-driven needs shape our mission. We rely on your feedback to provide the infrastructure you need most to advance your science, development, and educational agenda.

CyVerse Homepage: http://www.cyverse.org

Funding and Citations

CyVerse is funded entirely by the National Science Foundation under Award Numbers DBI-0735191 and DBI-1265383.

Please cite CyVerse appropriately when you make use of our resources, CyVerse citation policy

CHAPTER 5

Training session in Docker

Trainers (Kapeel Chougule and Upendra Devisetty)

In this session we will explain the various aspects of the Docker. Starting with the basics of Docker which focus on the installation and configuration of Docker, the session will gradually move to advanced topics such as managing data using volumes and pushing and pull containers from registries. Overall this session covers the development aspects of Docker and how you can get up and running on the development environments using Docker containers.

- Docker basics/Introduction (Kapeel)

This is the introductory session for the concept of Docker. The topics include Docker installation, running prebuilt Docker containers, deploying web applications with Docker, building and running your own Docker containers, etc.

- Advanced Docker (Upendra)

This is the advanced session for the concept of Docker. The topics include pushing and pulling Docker containers to public and private registries, automated Docker image building from github/bitbucket repositories, managing data in Docker containers, Docker compose for building multiple Docker containers, improving your data science workflows using Docker containers, etc.

CHAPTER 6

Training session in Singularity

Trainers (Tyson Swetnam and John Fonner)

In this session we will show you how to containerize your software/applications using Singularity, push them to Singularityhub and deploy them on cloud and HPC.

- Singularity basics/Introduction (Tyson Swetnam)

This would be the introductory session for concept of Singularity. The topics include installation Singularity on various platforms, running prebuilt singularity containers, building singularity containers locally etc.

- Advanced Singularity (John Fonner)

This is the advanced session for the concept of Singularity. The topics include pushing and pulling Singularity images to and from Singularity hub, converting Docker containers to Singularity containers, mounting data on to Singularity containers etc.

CHAPTER 7

Training session in scaling up your analysis using containers

Trainers (Mats Rynge, Nicholas Hazekamp and Kyle Sweeney)

In this exciting session of the workshop, we will show you how to scale your analyses (simple apps and complex workflow) using Docker swarm, Google Kubernetes and Workflows Management Systems such as Pegasus, Work-Queue and Makeflow from laptop to Cloud to HPC resources such as Stampede2, OSG and campus clusters and also show how using several compute clusters, you can scale your analysis significantly and efficiently. Some of the topics include:

- OSG (Open Science Grid) Singularity Infrastructure (Mats Rynge)
- Pegasus Workflows with Application Containers (Mats Rynge)
- Distributed computing with containers (Nicholas and Kyle **remotely**)

CHAPTER 8

Training session in Biocontainers

Trainers (Upendra Devisetty and John Fonner)

In this session we will show you how to containerize your bioinformatic software/applications (with special focus in Proteomics, Genomics, Transcriptomics and Metabolomics), push them to Dockerhub and other registries and finally deploy them on Cloud and HPC.

- Introduction to Biocontainers (Upendra Devisetty)

This would be the introductory session for concept of Biocontainers. The topics include what are biocontainers and how are they different from Docker containers, developing biocontainers, running Biocontainers, Biocontainer registry etc.

- Biocontainers in HPC environment (John Fonner)

This would be the session for concept of Biocontainers in HPC environment.

CHAPTER 9

Booting an Atmosphere computer instance for your use!

What we're going to do here is walk through of how to start up a running computer (an "instance") on the CyVerse Atmosphere Cloud service.

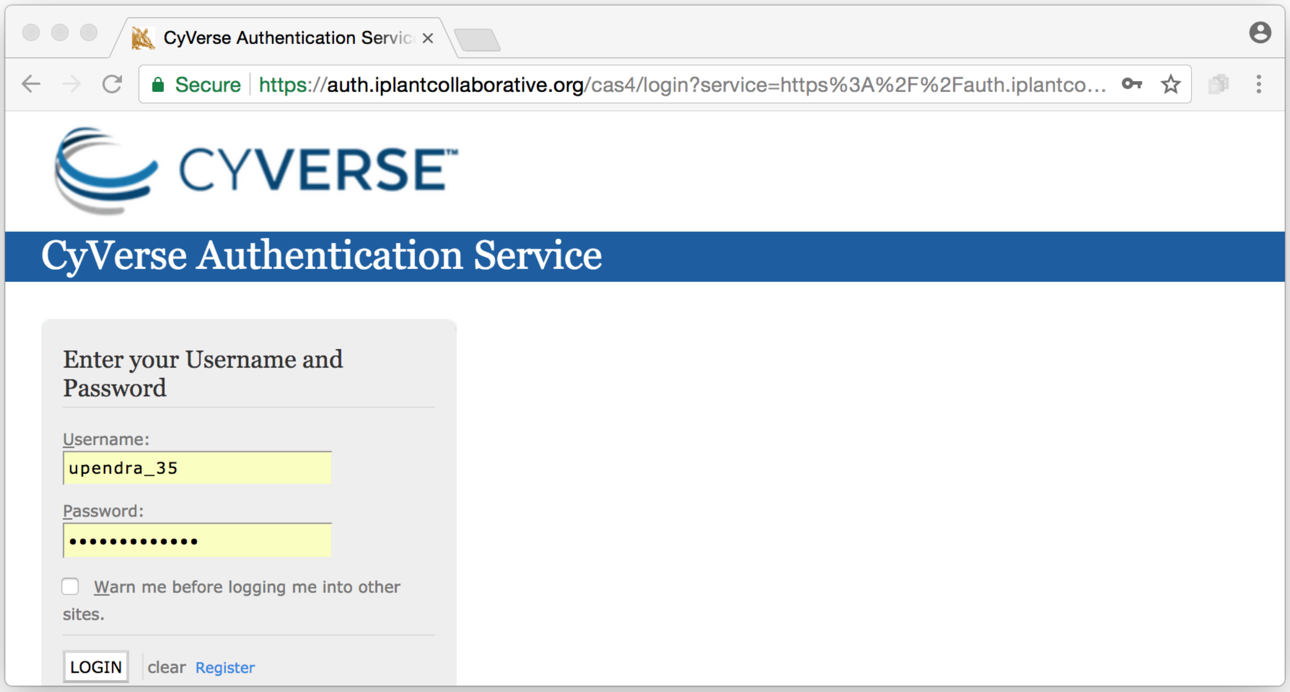
Below, we've provided screenshots of the whole process. You can click on them to zoom in a bit. The important areas to fill in are highlighted.

First, go to the Atmosphere application and then click *login*

Important: You will need to have access to the Atmosphere workshop cloud. If you are not able to log-in for some reason, please let us know and we will fix it immediately.

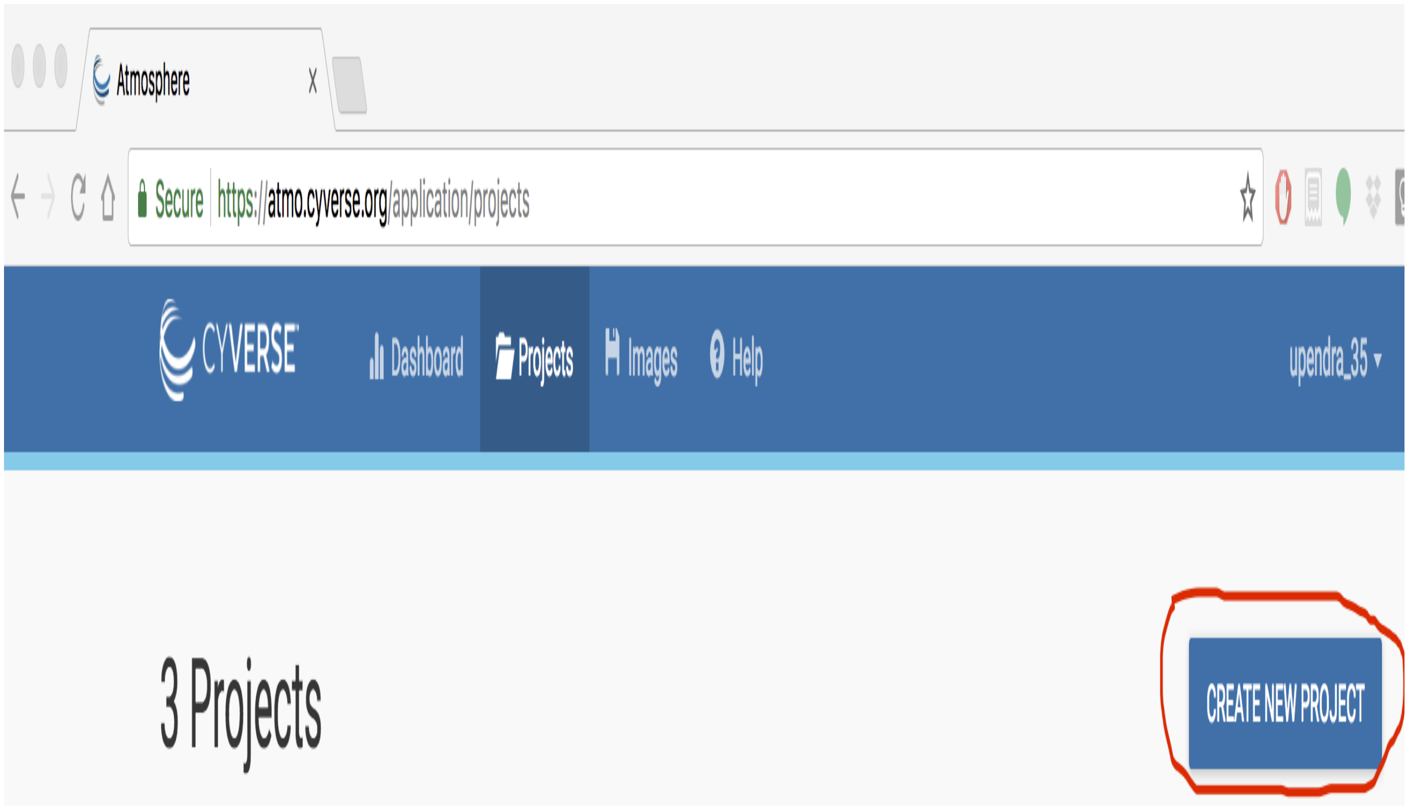
1. Fill in the username and password and click "LOGIN"

Fill in the username, which is your CyVerse username, and then enter the password (which is your CyVerse password).

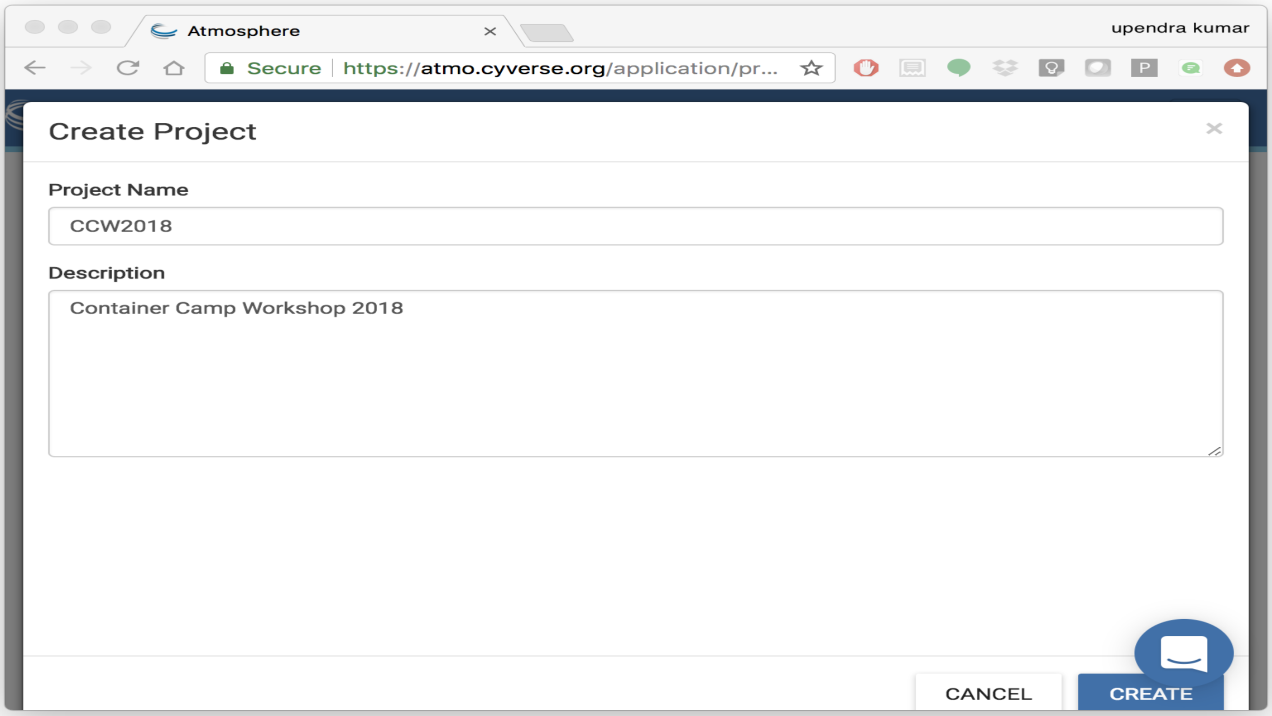

CyVerse Authentication Servic
Secure https://auth.iplantcollaborative.org/cas4/login?service=https%3A%2F%2Fauth.iplantco...
CYVERSE
CyVerse Authentication Service
Enter your Username and Password
Username:
upendra_35
Password:
•••••••••••••
Warn me before logging me into other sites.
LOGIN
clear
Register

2. Select Projects and "Create New Project"

- Now, this is something you only need to do once.
- We'll do this with Projects, which gives you a bit of a workspace in which to keep things that belong to "you".
- Click on the "Projects" tab on the top and then click "CREATE NEW PROJECT"



- Enter the name "CCW2018" into the Project Name, and something simple like "Container Camp Workshop 2018" into the description. Then click 'create'.



Create Project

Project Name

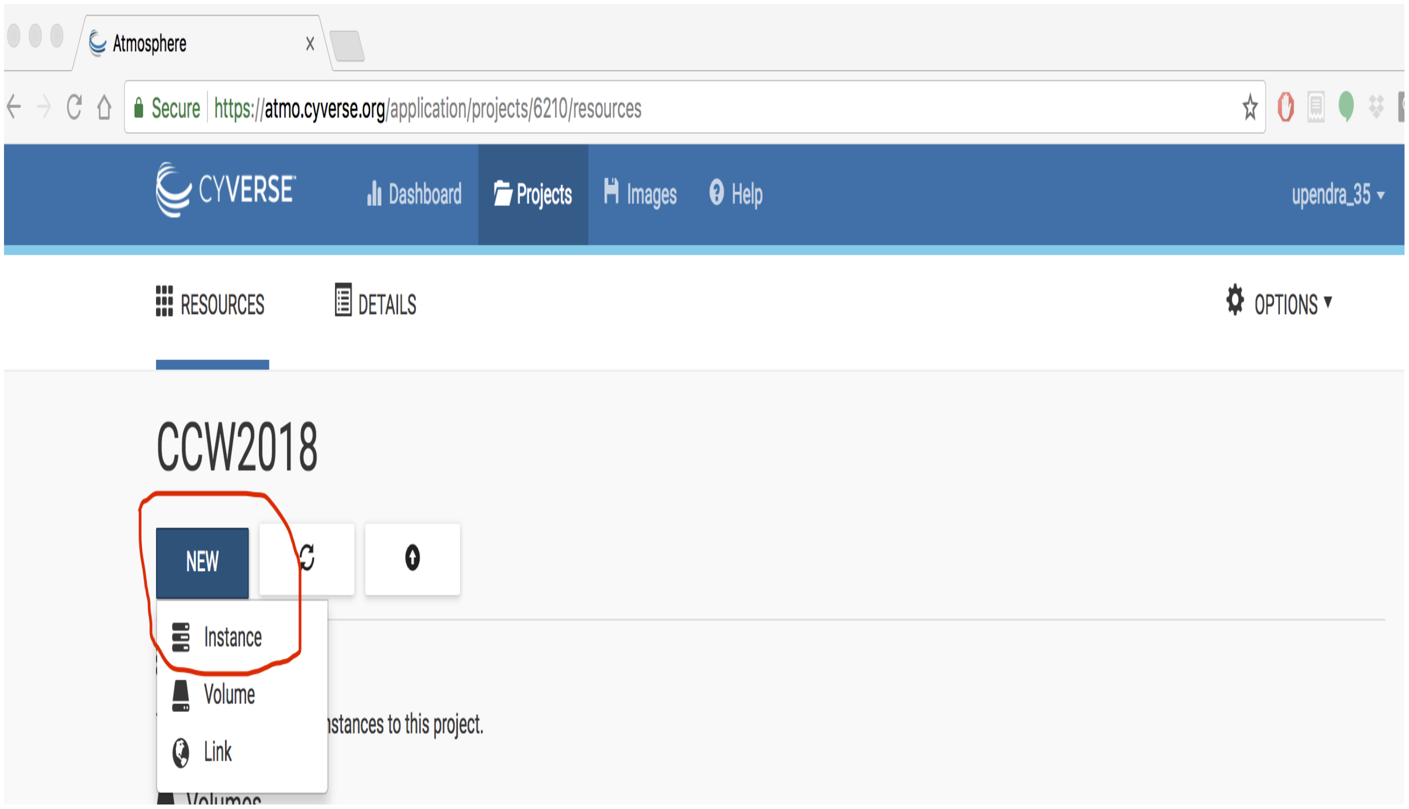
CCW2018

Description

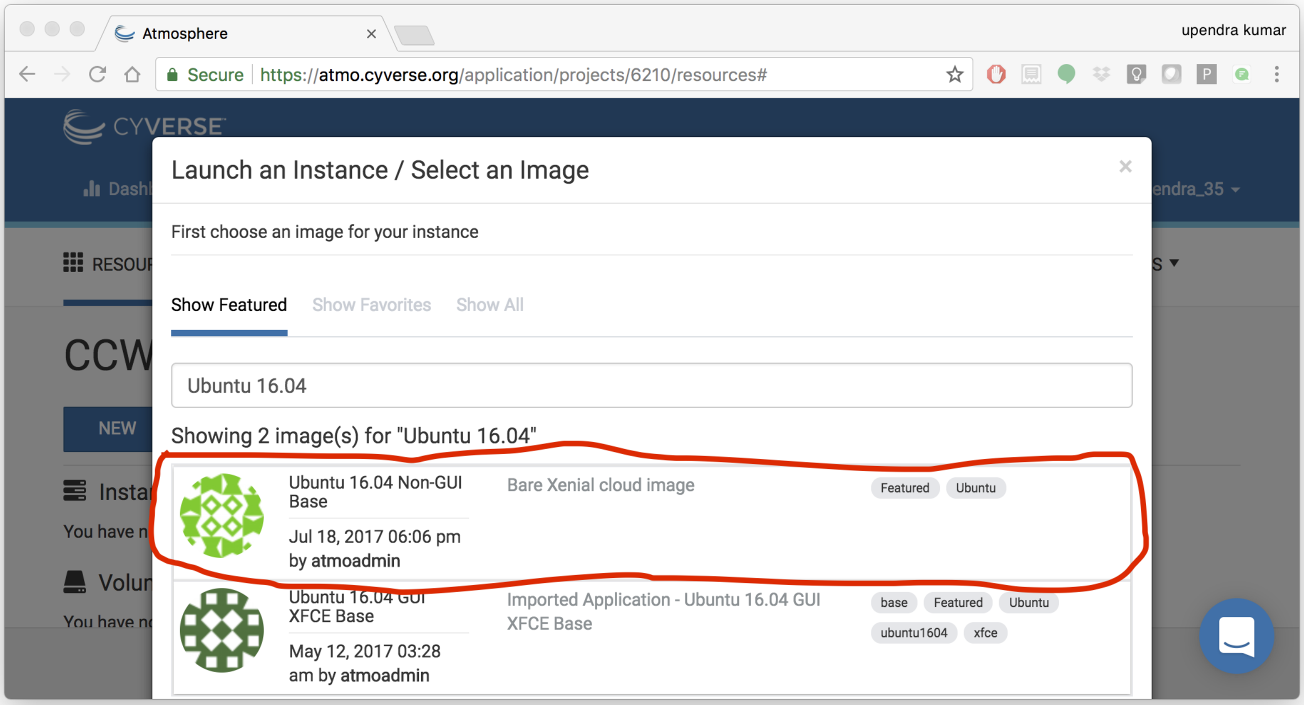
Container Camp Workshop 2018

3. Select the newly created project

- Click on your newly created project!
- Now, click 'New' and then "Instance" from the dropdown menu to start up a new virtual machine.



- Find the "Ubuntu 16.04" image, click on it



Launch an Instance / Select an Image

First choose an image for your instance

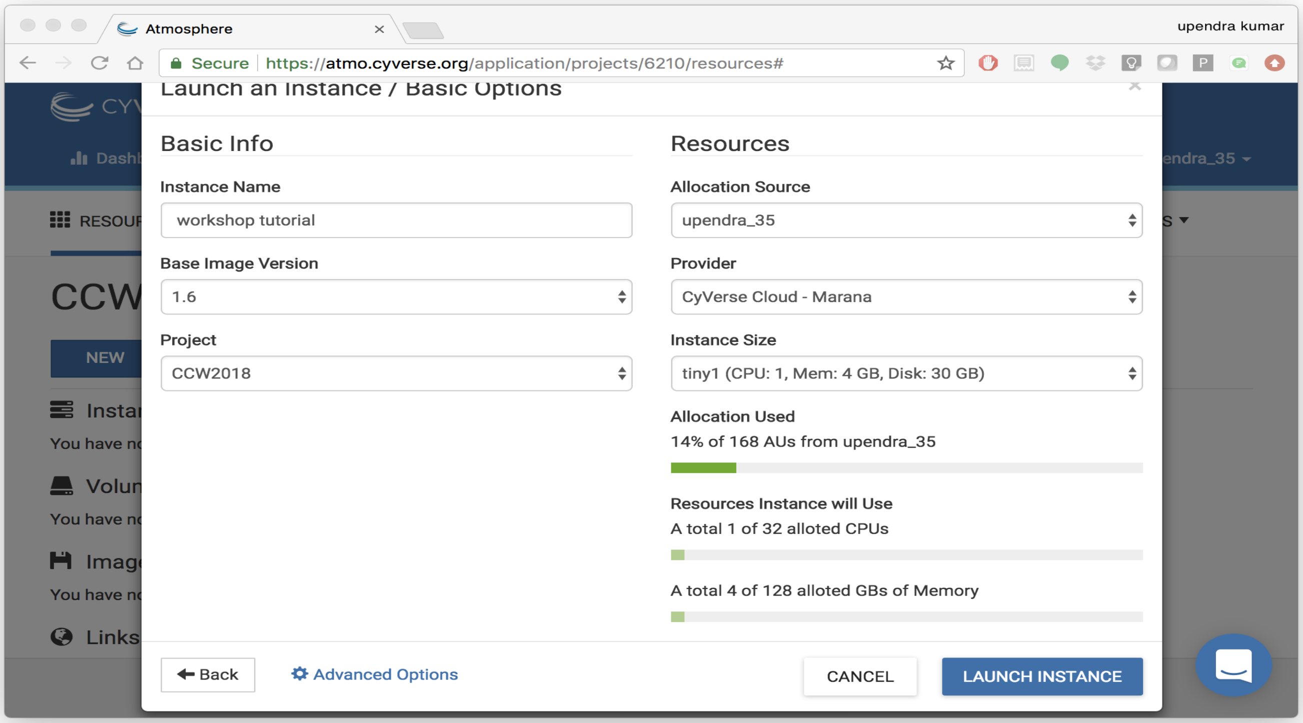
Show Featured | Show Favorites | Show All

Ubuntu 16.04

Showing 2 image(s) for "Ubuntu 16.04"

Ubuntu 16.04 Non-GUI Base	Jul 18, 2017 06:06 pm by atmoadmin	Bare Xenial cloud image	Featured, Ubuntu
Ubuntu 16.04 GUI XFCE Base	May 12, 2017 03:28 am by atmoadmin	Imported Application - Ubuntu 16.04 GUI XFCE Base	base, Featured, Ubuntu, ubuntu1604, xfce

- Name it something simple such as "workshop tutorial" and select 'tiny1 (CPU: 1, Mem: 4GB, Disk: 30GB)'.
- Leave rest of the fields as default.



Launch an Instance / Basic Options

Basic Info

Instance Name: workshop tutorial

Base Image Version: 1.6

Project: CCW2018

Resources

Allocation Source: upendra_35

Provider: CyVerse Cloud - Marana

Instance Size: tiny1 (CPU: 1, Mem: 4 GB, Disk: 30 GB)

Allocation Used

14% of 168 AUs from upendra_35

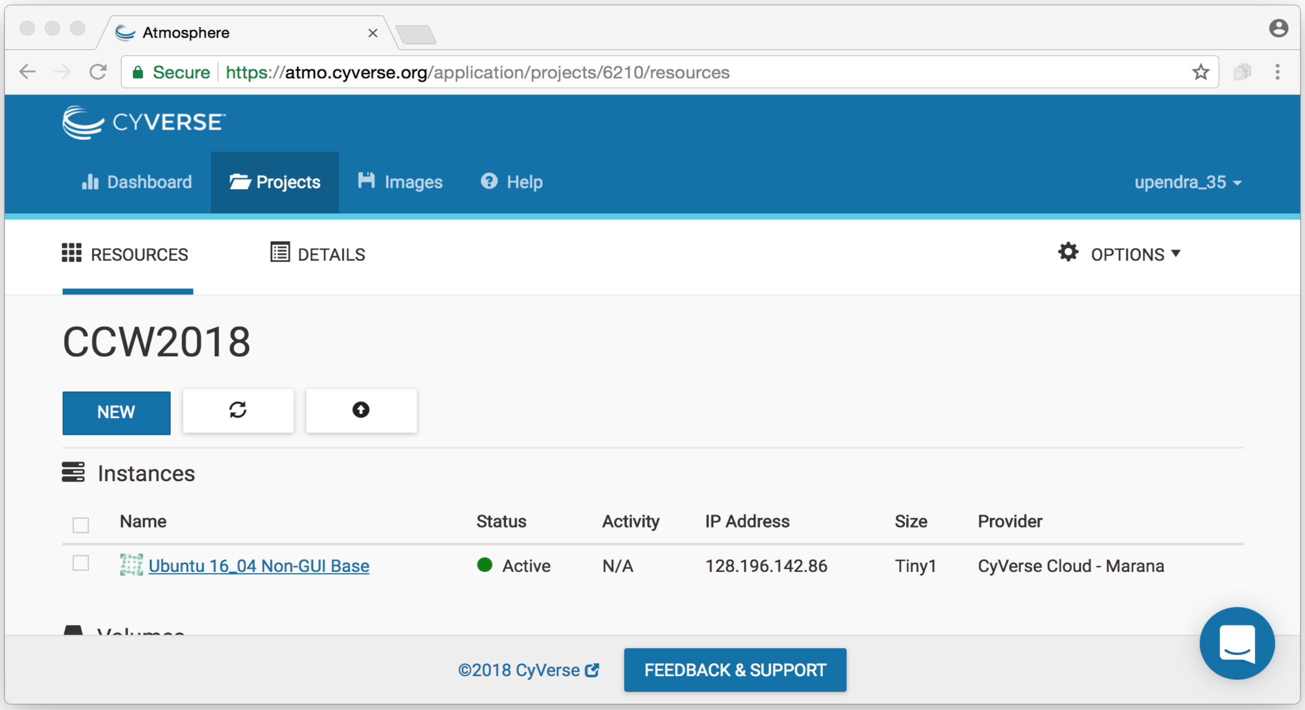
Resources Instance will Use

A total 1 of 32 alloted CPUs

A total 4 of 128 alloted GBs of Memory

- Wait for it to become active
- It will now be booting up! This will take 2-10 minutes, depending.

Just wait! Don't reload or anything.



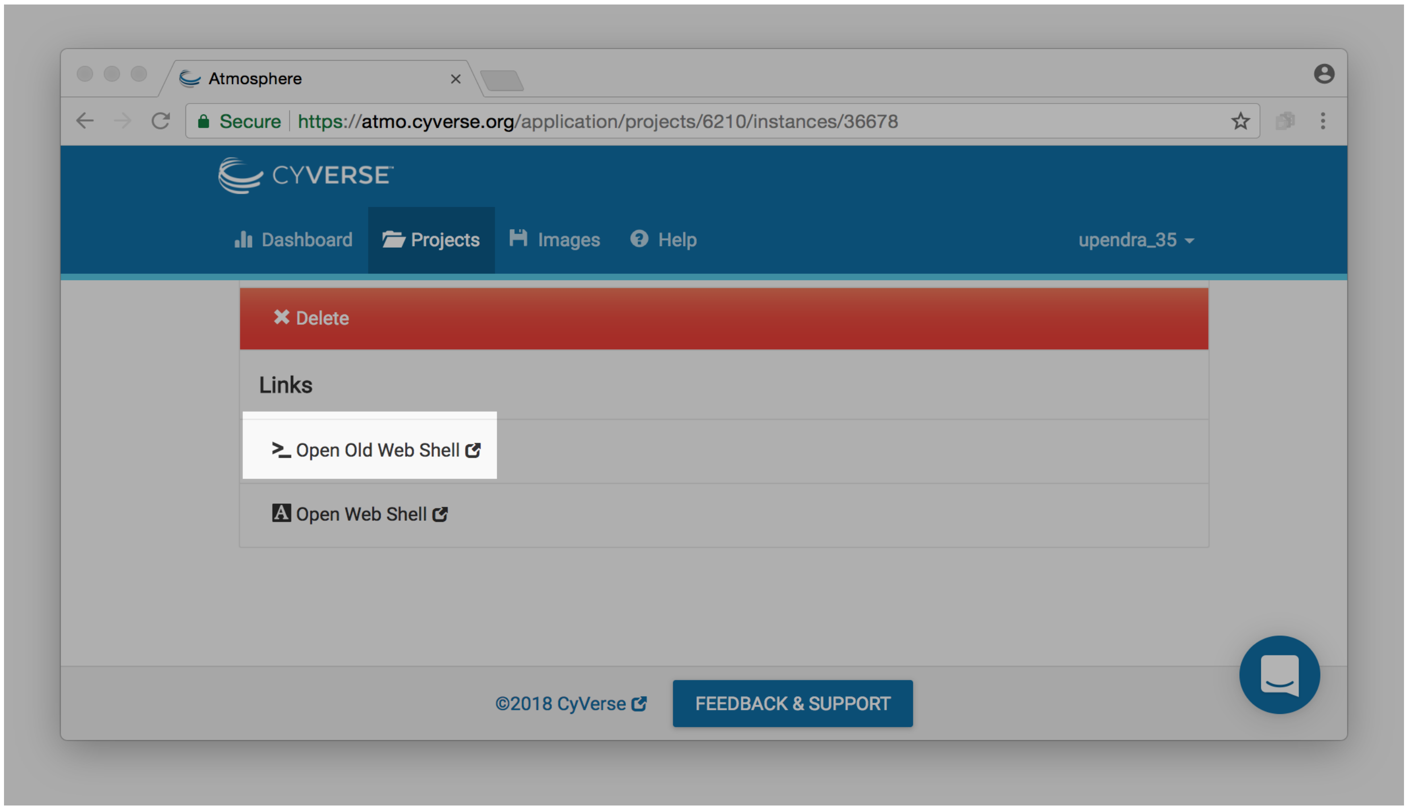
CCW2018

Instances

Name	Status	Activity	IP Address	Size	Provider
Ubuntu 16_04 Non-GUI Base	● Active	N/A	128.196.142.86	Tiny1	CyVerse Cloud - Marana

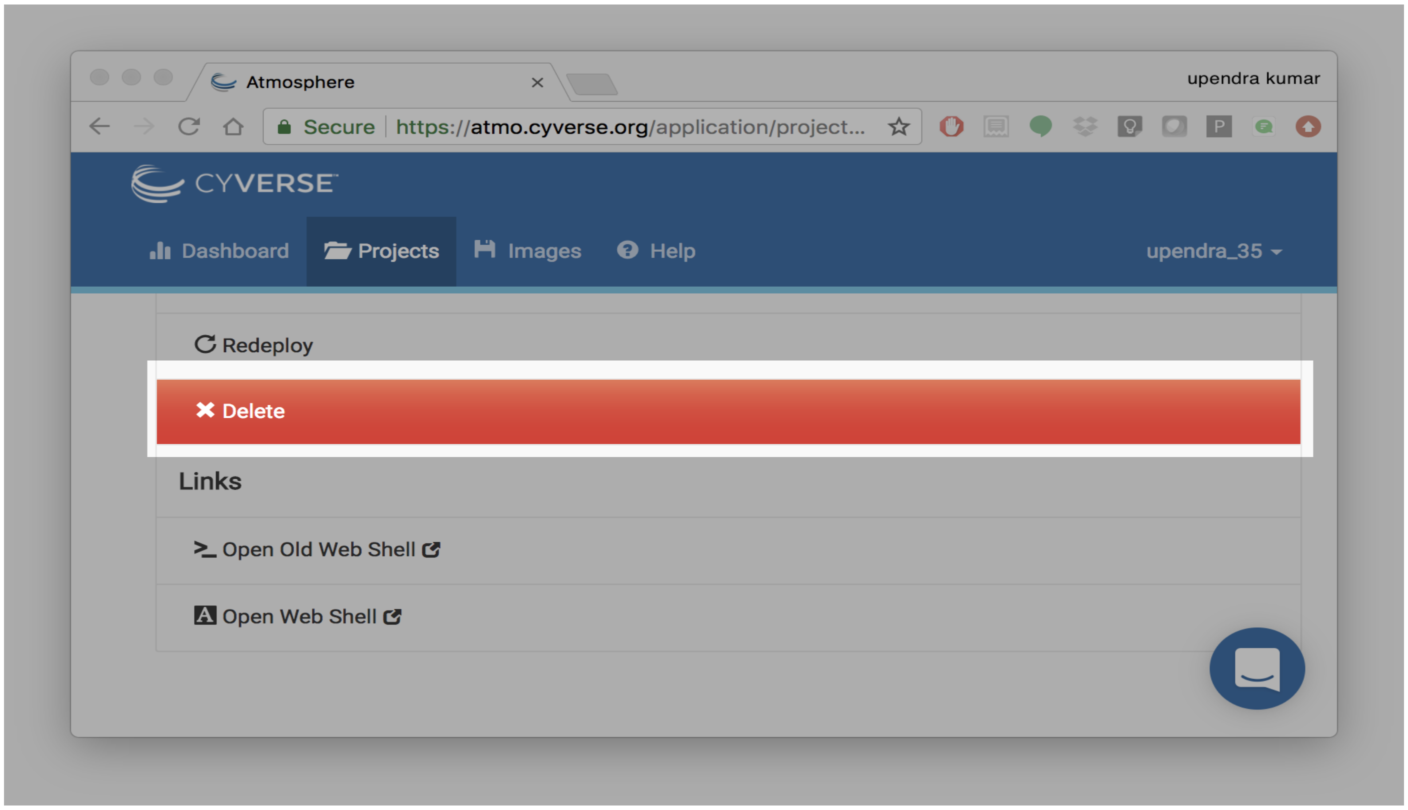
- Click on your new instance to get more information!
- Now, you can either click "Open Web Shell", *or*, if you know how to use ssh,

you can ssh in with your CyVerse username on the IP address of the machine

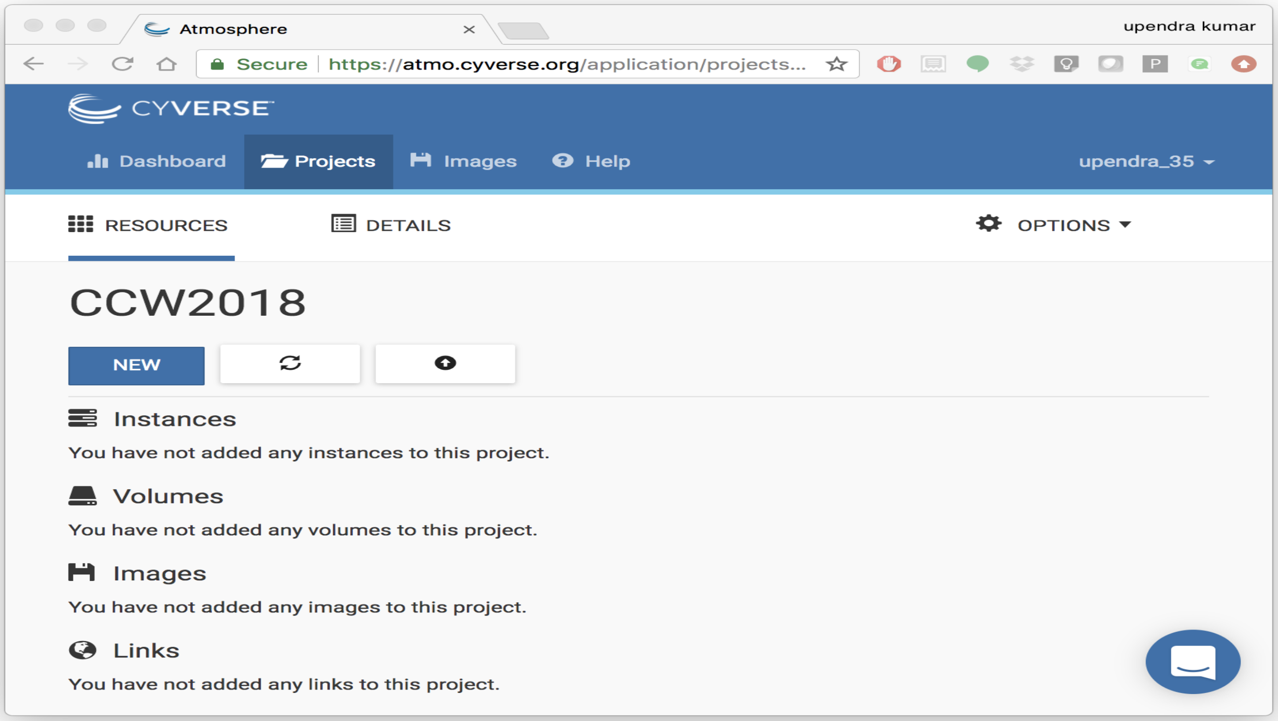


Deleting your instance

- To completely remove your instance, you can select the "delete" buttom from the instance details page.
- This will open up a dialogue window. Select the "Yes, delete this instance" button.



- It may take Atmosphere a few minutes to process your request. The instance should disappear from the project when it has been successfully deleted.



You have not added any instances to this project.

You have not added any volumes to this project.

You have not added any images to this project.

You have not added any links to this project.

Note: It is advisable to delete the machine if you are not planning to use it in future to save valuable resources. However if you want to use it in future, you can suspend it.

CHAPTER 10

Introduction to Docker

10.1 1. Prerequisites

There are no specific skills needed for this tutorial beyond a basic comfort with the command line and using a text editor. Prior experience in developing web applications will be helpful but is not required.

10.2 2. Docker Installation

Getting all the tooling setup on your computer can be a daunting task, but not with Docker. Getting Docker up and running on your favorite OS (Mac/Windows/Linux) is very easy.

The getting started guide on Docker has detailed instructions for setting up Docker on Mac/Windows/Linux.

Note: If you're using Docker for Windows make sure you have shared your drive.

If you're using an older version of Windows or MacOS you may need to use Docker Machine instead.

All commands work in either bash or Powershell on Windows.

Note: Depending on how you've installed Docker on your system, you might see a `permission denied` error after running the above command. If you're on Linux, you may need to prefix your Docker commands with sudo. Alternatively to run docker command without sudo, you need to add your user (who has root privileges) to docker group. For this run:

Create the docker group:

```
$ sudo groupadd docker
```

Add your user to the docker group:

```
$ sudo usermod -aG docker $USER
```

Log out and log back in so that your group membership is re-evaluated

10.2.1 2.1 Testing Docker installation

Once you are done installing Docker, test your Docker installation by running the following command to make sure you are using version 1.13 or higher:

```
$ docker --version
Docker version 17.09.0-ce, build afdb6d4
```

When run without `--version` you should see a whole bunch of lines showing the different options available with `docker`. Alternatively you can test your installation by running the following:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest
```

(continues on next page)

(continued from previous page)

```

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
.......
```

10.3 3. Running Docker containers from prebuilt images

Now that you have everything setup, it's time to get our hands dirty. In this section, you are going to run a container from Alpine Linux (a lightweight linux distribution) image on your system and get a taste of the `docker run` command.

But wait, what exactly is a container and image?

Containers - Running instances of Docker images — containers run the actual applications. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS.

Images - The file system and configuration of our application which are used to create containers. To find out more about a Docker image, run `docker inspect hello-world`. In the demo above, you could have used the `docker pull` command to download the `hello-world` image. However when you executed the command `docker run hello-world`, it also did a `docker pull` behind the scenes to download the `hello-world` image with `latest` tag (we will learn more about tags little later).

Now that we know what a container and image is, let's run the following command in our terminal:

```
$ docker run alpine ls -l
total 52
drwxr-xr-x    2 root     root          4096 Dec 26  2016 bin
drwxr-xr-x    5 root     root           340 Jan 28 09:52 dev
drwxr-xr-x   14 root     root          4096 Jan 28 09:52 etc
drwxr-xr-x    2 root     root          4096 Dec 26  2016 home
drwxr-xr-x    5 root     root          4096 Dec 26  2016 lib
drwxr-xr-x    5 root     root          4096 Dec 26  2016 media
........
```

Similar to `docker run hello-world` command in the demo above, `docker run alpine ls -l` command fetches the `alpine:latest` image from the Docker registry first, saves it in our system and then runs a container from that saved image.

When you run `docker run alpine`, you provided a command `ls -l`, so Docker started the command specified and you saw the listing

You can use the `docker images` command to see a list of all images on your system

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             ␣
↪VIRTUAL SIZE
```

(continues on next page)

(continued from previous page)

```
alpine                  latest              c51f86c28340        4 weeks ago         1.
↪109 MB
hello-world             latest              690ed74de00f        5 months ago        ␣
↪960 B
```

Let's try something more exciting.

```
$ docker run alpine echo "Hello world"
Hello world
```

OK, that's some actual output. In this case, the Docker client dutifully ran the `echo` command in our `alpine` container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast!

Try another command.

```
$ docker run alpine sh
```

Wait, nothing happened! Is that a bug? Well, no. These interactive shells will exit after running any scripted commands such as `sh`, unless they are run in an interactive terminal - so for this example to not exit, you need to `docker run -it alpine sh`. You are now inside the container shell and you can try out a few commands like `ls -l`, `uname -a` and others.

Before doing that, now it's time to see the `docker ps` command which shows you all containers that are currently running.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             ␣
↪STATUS              PORTS               NAMES
```

Since no containers are running, you see a blank line. Let's try a more useful variant: `docker ps -a`

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             ␣
↪STATUS                      PORTS               NAMES
36171a5da744        alpine               "/bin/sh"               5 minutes ago       ␣
↪Exited (0) 2 minutes ago                        fervent_newton
a6a9d46d0b2f        alpine              "echo 'hello from alp"   6 minutes ago       ␣
↪Exited (0) 6 minutes ago                        lonely_kilby
ff0a5c3750b9        alpine              "ls -l"                  8 minutes ago       ␣
↪Exited (0) 8 minutes ago                        elated_ramanujan
c317d0a9e3d2        hello-world          "/hello"                34 seconds ago      ␣
↪Exited (0) 12 minutes ago                       stupefied_mcclintock
```

What you see above is a list of all containers that you ran. Notice that the STATUS column shows that these containers exited a few minutes ago.

If you want to run scripted commands such as `sh`, they should be run in an interactive terminal. In addition, interactive terminal allows you to run more than one command in a container. Let's try that now:

```
$ docker run -it alpine sh
/ # ls
bin      dev      etc      home     lib      media    mnt      proc     root     run      sbin     srv      ␣
↪sys      tmp      usr      var
/ # uname -a
Linux de4bbc3eeaec 4.9.49-moby #1 SMP Wed Sep 27 23:17:17 UTC 2017 x86_64 Linux
```

Running the `run` command with the `-it` flags attaches us to an interactive `tty` in the container. Now you can run as many commands in the container as you want. Take some time to run your favorite commands.

Exit out of the container by giving the `exit` command.

```
/ # exit
```

Note: If you type `exit` your **container** will exit and is no longer active. To check that, try the following:

```
$ docker ps -l
CONTAINER ID        IMAGE                 COMMAND                  CREATED        ␣
↪ STATUS                        PORTS                    NAMES
de4bbc3eeaec        alpine                "/bin/sh"                3 minutes ago  ␣
↪ Exited (0) About a minute ago                          pensive_leavitt
```

If you want to keep the container active, then you can use keys `ctrl +p, ctrl +q`. To make sure that it is not exited run the same `docker ps -a` command again:

```
$ docker ps -l
CONTAINER ID        IMAGE                 COMMAND                  CREATED        ␣
↪ STATUS                       PORTS                     NAMES
0db38ea51a48        alpine                "sh"                     3 minutes ago  ␣
↪ Up 3 minutes                                           elastic_lewin
```

Now if you want to get back into that container, then you can type `docker attach <container id>`. This way you can save your container:

```
$ docker attach 0db38ea51a48
```

10.4 4. Deploying web applications with Docker

Great! so you have now looked at `docker run`, played with a Docker containers and also got the hang of some terminology. Armed with all this knowledge, you are now ready to get to the real stuff — deploying web applications with Docker.

10.4.1 4.1 Deploying static website

Let's start by taking baby-steps. First, we'll use Docker to run a static website in a container. The website is based on an existing image and in the next section we will see how to build a new image and run a website in that container. We'll pull a Docker image from Dockerhub, run the container, and see how easy it is to set up a web server.

Note: Code for this section is in this repo in the static-site directory

The image that you are going to use is a single-page website that was already created for this demo and is available on the Dockerhub as dockersamples/static-site. You can pull and run the image directly in one go using `docker run` as follows.

```
$ docker run -d dockersamples/static-site
```

Note: The `-d` flag enables detached mode, which detaches the running container from the terminal/shell and returns your prompt after the container starts.

So, what happens when you run this command?

Since the image doesn't exist on your Docker host (laptop/computer), the Docker daemon first fetches it from the registry and then runs it as a container.

Now that the server is running, do you see the website? What port is it running on? And more importantly, how do you access the container directly from our host machine?

Actually, you probably won't be able to answer any of these questions yet! In this case, the client didn't tell the Docker Engine to publish any of the ports, so you need to re-run the `docker run` command to add this instruction.

Let's re-run the command with some new flags to publish ports and pass your name to the container to customize the message displayed. We'll use the `-d` option again to run the container in detached mode.

First, stop the container that you have just launched. In order to do this, we need the container ID.

Since we ran the container in detached mode, we don't have to launch another terminal to do this. Run `docker ps` to view the running containers.

```
$ docker ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED          ␣
↪   STATUS              PORTS                  NAMES
a7a0e504ca3e        dockersamples/static-site   "/bin/sh -c 'cd /usr/"   28 seconds␣
↪ago       Up 26 seconds        80/tcp, 443/tcp     stupefied_mahavira
```

Check out the CONTAINER ID column. You will need to use this CONTAINER ID value, a long sequence of characters, to identify the container you want to stop, and then to remove it. The example below provides the CONTAINER ID on our system; you should use the value that you see in your terminal.

```
$ docker stop a7a0e504ca3e
$ docker rm   a7a0e504ca3e
```

Note: A cool feature is that you do not need to specify the entire **CONTAINER ID**. You can just specify a few starting characters and if it is unique among all the containers that you have launched, the Docker client will intelligently pick it up.

Now, let's launch a container in detached mode as shown below:

```
$ docker run --name static-site -d -P dockersamples/static-site
e61d12292d69556eabe2a44c16cbd54486b2527e2ce4f95438e504afb7b02810
```

In the above command:

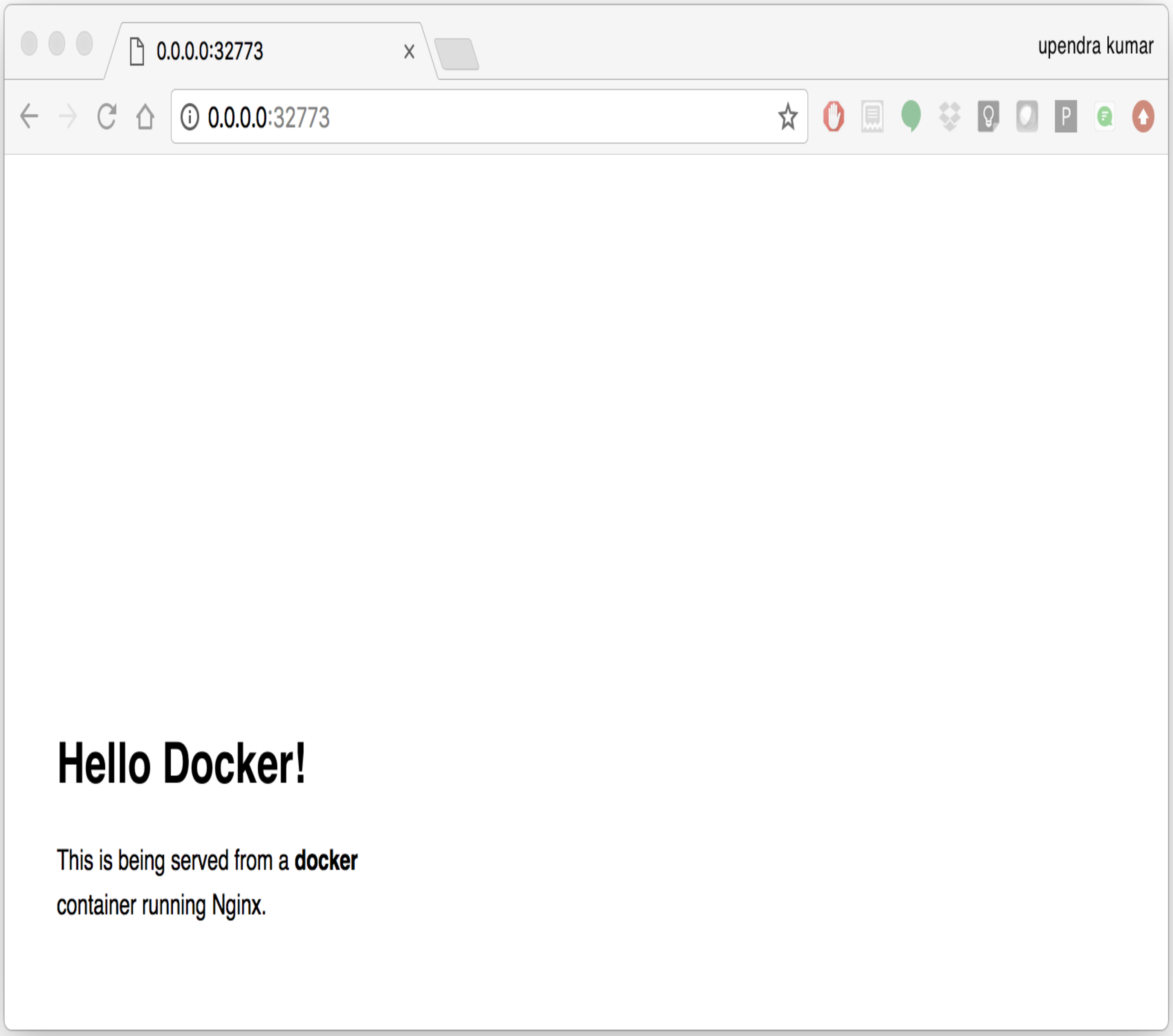
- `-d` will create a container with the process detached from our terminal
- `-P` will publish all the exposed container ports to random ports on the Docker host
- `--name` allows you to specify a container name

Now you can see the ports by running the `docker port` command.

```
$ docker port static-site
443/tcp -> 0.0.0.0:32770
80/tcp -> 0.0.0.0:32773
```

If you are running Docker for Mac, Docker for Windows, or Docker on Linux, open a web browser and go to port 80 on your host. The exact address will depend on how you're running Docker

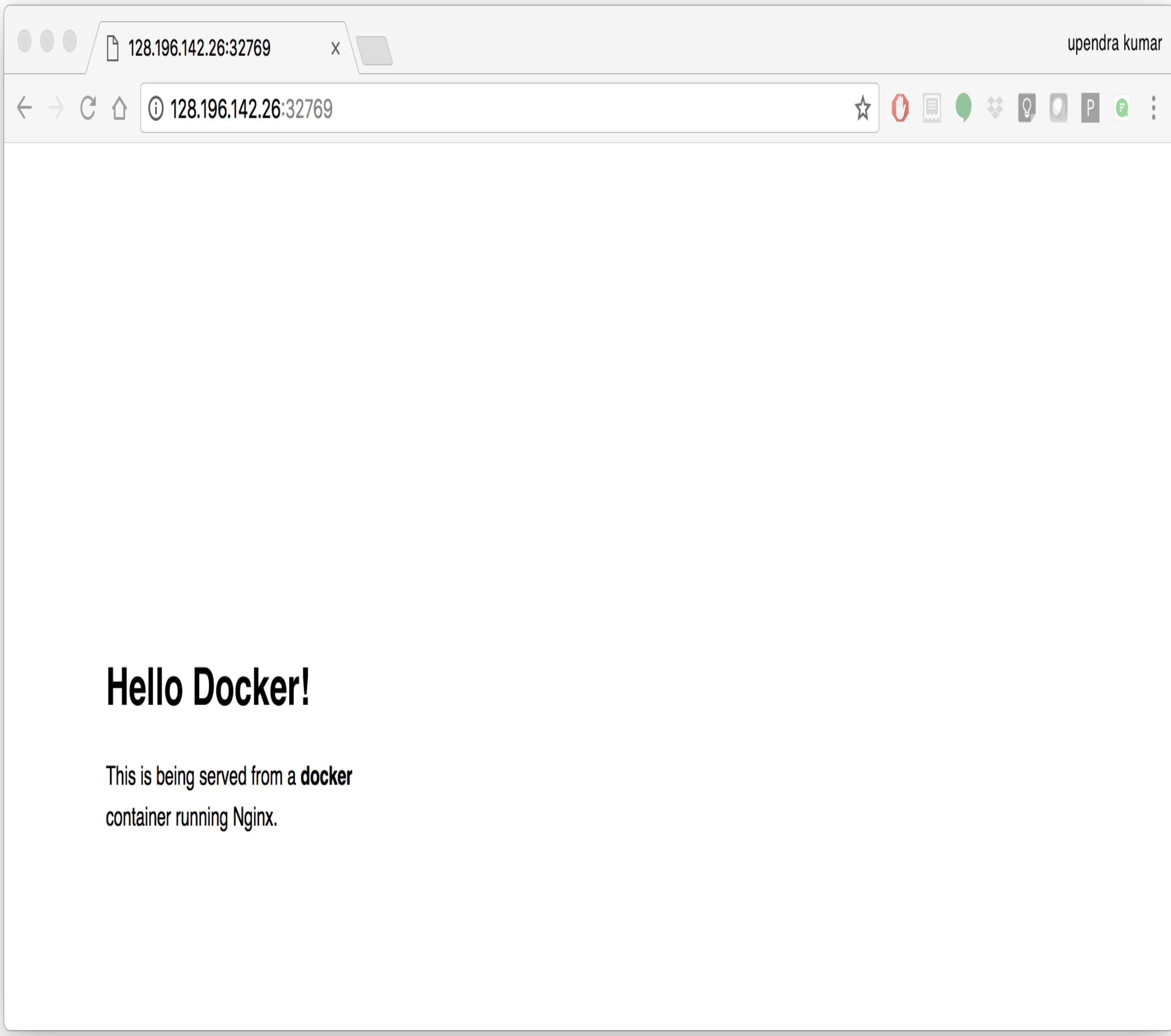
- Laptop or Native linux: `http://localhost:[YOUR_PORT_FOR 80/tcp]`. On my system this is `http://localhost:32773`.



Hello Docker!

This is being served from a **docker** container running Nginx.

- Cloud server: If you are running the same set of commands on Atmosphere/Jetstream or on any other cloud service, you can open `ipaddress:[YOUR_PORT_FOR 80/tcp]`. On my Atmosphere instance this is `http://128.196.142.26:32769/`. We will see more about deploying Docker containers on Atmosphere/Jetstream Cloud in the Advanced Docker session.



Hello Docker!

This is being served from a **docker** container running Nginx.

Note: `-P`` `will publish all the exposed container ports to random ports on the Docker host. However if you want to assign a fixed port then you can use ``-p` option. The format is `-p <host port>:<container port>`. For example:

```
$ docker run --name static-site2 -d -p 8088:80 dockersamples/static-site
```

If you are running Docker for Mac, Docker for Windows, or Docker on Linux, you can open `http://localhost:[YOUR_PORT_FOR 80/tcp]`. For our example this is `http://localhost:8088`.

If you are running Docker on Atmosphere/Jetstream or on any other cloud, you can open `ipaddress:[YOUR_PORT_FOR 80/tcp]`. For our example this is `http://128.196.142.26:8088/`

If you see "Hello Docker!" then you're done!

Let's stop and remove the containers since you won't be using them anymore.

```
$ docker stop static-site static-site2
$ docker rm static-site static-site2
```

Let's use a shortcut to both stop and delete that container from your system:

```
$ docker rm -f static-site static-site2
```

Run `docker ps` to make sure the containers are gone.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             ␣
↪STATUS              PORTS               NAMES
```

10.4.2 4.2 Deploying dynamic website

One area where Docker shines is when you need to use a command line utility that has a large number of dependencies.

In this section, let's dive deeper into what Docker images are. Later on we will build our own image and use that image to run an application locally (deploy a dynamic website).

4.2.1 Docker images

Docker images are the basis of containers. In the previous example, you pulled the `dockersamples/static-site` image from the registry and asked the Docker client to run a container based on that image. To see the list of images that are available locally on your system, run the `docker images` command.

```
$ docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED         ␣
↪       SIZE
dockersamples/static-site   latest              92a386b6e686        2 hours ago         ␣
↪ 190.5 MB
nginx                           latest              af4b3d7d5401        3 hours ago     ␣
↪     190.5 MB
hello-world                     latest              690ed74de00f        5 months ago    ␣
↪     960 B
.........
```

Above is a list of images that I've pulled from the registry and those I've created myself (we'll shortly see how). You will have a different list of images on your machine. The **TAG** refers to a particular snapshot of the image and the **ID** is the corresponding unique identifier for that image.

For simplicity, you can think of an image akin to a git repository - images can be committed with changes and have multiple versions. When you do not provide a specific version number, the client defaults to latest.

For example you could pull a specific version of ubuntu image as follows:

```
$ docker pull ubuntu:16.04
```

If you do not specify the version number of the image, as mentioned, the Docker client will default to a version named `latest`.

So for example, the `docker pull` command given below will pull an image named `ubuntu:latest`

```
$ docker pull ubuntu
```

To get a new Docker image you can either get it from a registry (such as the Docker hub) or create your own. There are hundreds of thousands of images available on Docker hub. You can also search for images directly from the command line using `docker search`.

```
$ docker search ubuntu
 NAME                                                        DESCRIPTION                                  ␣
↪                    STARS               OFFICIAL            AUTOMATED
 ubuntu                                                      Ubuntu is a Debian-based␣
↪Linux operating sys...   7310                      [OK]
 dorowu/ubuntu-desktop-lxde-vnc                              Ubuntu with openssh-server␣
↪and NoVNC               163                                       [OK]
 rastasheep/ubuntu-sshd                                      Dockerized SSH service,␣
↪built on top of offi...   131                                          [OK]
 ansible/ubuntu14.04-ansible                                 Ubuntu 14.04 LTS with␣
↪ansible                    90                                          [OK]
 ubuntu-upstart                                              Upstart is an event-based␣
↪replacement for th...   81                        [OK]
 neurodebian                                                 NeuroDebian provides␣
↪neuroscience research s...   43                        [OK]
 ubuntu-debootstrap                                          debootstrap --
↪variant=minbase --components=m...   35                    [OK]
 1and1internet/ubuntu-16-nginx-php-phpmyadmin-mysql-5   ubuntu-16-nginx-php-
↪phpmyadmin-mysql-5            26                                          [OK]
 nuagebec/ubuntu                                             Simple always updated Ubuntu␣
↪docker images w...   22                                       [OK]
 tutum/ubuntu                                                Simple Ubuntu docker images␣
↪with SSH access      18
 ppc64le/ubuntu                                              Ubuntu is a Debian-based␣
↪Linux operating sys...   11
 i386/ubuntu                                                 Ubuntu is a Debian-based␣
↪Linux operating sys...   9
 1and1internet/ubuntu-16-apache-php-7.0                      ubuntu-16-apache-php-7.0     ␣
↪                    7                                       [OK]
 eclipse/ubuntu_jdk8                                         Ubuntu, JDK8, Maven 3, git,␣
↪curl, nmap, mc, ...    5                                       [OK]
 darksheer/ubuntu                                            Base Ubuntu Image -- Updated␣
↪hourly              3                                       [OK]
 codenvy/ubuntu_jdk8                                         Ubuntu, JDK8, Maven 3, git,␣
↪curl, nmap, mc, ...    3                                       [OK]
 1and1internet/ubuntu-16-nginx-php-5.6-wordpress-4       ubuntu-16-nginx-php-5.6-
↪wordpress-4               2                                       [OK]
 1and1internet/ubuntu-16-nginx                               ubuntu-16-nginx              ␣
↪                    2                                       [OK]
 pivotaldata/ubuntu                                          A quick freshening-up of the␣
↪base Ubuntu doc...   1
 smartentry/ubuntu                                           ubuntu with smartentry       ␣
↪                    0                                       [OK]
 pivotaldata/ubuntu-gpdb-dev                                 Ubuntu images for GPDB␣
↪development               0
 1and1internet/ubuntu-16-healthcheck                         ubuntu-16-healthcheck        ␣
↪                    0                                       [OK]
 thatsamguy/ubuntu-build-image                               Docker webapp build images␣
↪based on Ubuntu       0
 ossobv/ubuntu                                               Custom ubuntu image from␣
↪scratch (based on o...   0
```

(continues on next page)

(continued from previous page)

```
 1and1internet/ubuntu-16-sshd                                   ubuntu-16-sshd                 ␣
↪                    0                                              [OK]
```

An important distinction with regard to images is between base images and child images and official images and user images (Both of which can be base images or child images.).

Important: **Base images** are images that have no parent images, usually images with an OS like ubuntu, alpine or debian.

Child images are images that build on base images and add additional functionality.

Official images are Docker sanctioned images. Docker, Inc. sponsors a dedicated team that is responsible for reviewing and publishing all Official Repositories content. This team works in collaboration with upstream software maintainers, security experts, and the broader Docker community. These are not prefixed by an organization or user name. In the list of images above, the python, node, alpine and nginx images are official (base) images. To find out more about them, check out the Official Images Documentation.

User images are images created and shared by users like you. They build on base images and add additional functionality. Typically these are formatted as `user/image-name`. The user value in the image name is your Dockerhub user or organization name.

4.2.2 Meet our Flask app

Now that you have a better understanding of images, it's time to create an image that sandboxes a small Flask application. Flask is a lightweight Python web framework. We'll do this by first pulling together the components for a random cat picture generator built with Python Flask, then dockerizing it by writing a Dockerfile and finally we'll build the image and run it.

- *Create a Python Flask app that displays random cat*
- *Build the image*
- *Run your image*

Note: I have already written the Flask app for you, so you should start by cloning the git repository at https://github.com/upendrak/flask-app. You can do this with `git clone` if you have git installed, or by clicking the "Download ZIP" button on GitHub

1. Create a Python Flask app that displays random cat

For the purposes of this workshop, we've created a fun little Python Flask app that displays a random cat .gif every time it is loaded - because, you know, who doesn't like cats?

Start by creating a directory called `flask-app` where we'll create the following files:

- *app.py*
- *requirements.txt*
- *templates/index.html*
- *Dockerfile*

```
$ mkdir flask-app && cd flask-app
```

1.1 **app.py**

Create the `app.py` file with the following content. You can use any of favorite text editor to create this file.

```
from flask import Flask, render_template
import random

app = Flask(__name__)

# list of cat images
images = [
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-
↪buzz-26388-1381844103-11.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr01/15/9/anigif_enhanced-
↪buzz-31540-1381844535-8.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-
↪buzz-26390-1381844163-18.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/10/anigif_enhanced-
↪buzz-1376-1381846217-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/9/anigif_enhanced-
↪buzz-3391-1381844336-26.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/10/anigif_enhanced-
↪buzz-29111-1381845968-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/9/anigif_enhanced-
↪buzz-3409-1381844582-13.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr02/15/9/anigif_enhanced-
↪buzz-19667-1381844937-10.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-
↪buzz-26358-1381845043-13.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/9/anigif_enhanced-
↪buzz-18774-1381844645-6.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/9/anigif_enhanced-
↪buzz-25158-1381844793-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/10/anigif_enhanced-
↪buzz-11980-1381846269-1.gif"
]

@app.route('/')
def index():
    url = random.choice(images)
    return render_template('index.html', url=url)

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

1.2. **requirements.txt**

In order to install the Python modules required for our app, we need to create a file called `requirements.txt` and add the following line to that file:

```
Flask==0.10.1
```

1.3. **templates/index.html**

Create a directory called *templates* and create an `index.html` file in that directory with the following content in it:

```
$ mkdir templates && cd templates
```

```
<html>
  <head>
    <style type="text/css">
      body {
        background: black;
        color: white;
      }
      div.container {
        max-width: 500px;
        margin: 100px auto;
        border: 20px solid white;
        padding: 10px;
        text-align: center;
      }
      h4 {
        text-transform: uppercase;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <h4>Cat Gif of the day</h4>
      <img src="{{url}}" />
      <p><small>Courtesy: <a href="http://www.buzzfeed.com/copyranter/the-best-cat-
↪gif-post-in-the-history-of-cat-gifs">Buzzfeed</a></small></p>
    </div>
  </body>
</html>
```

Note: If you want, you can run this app through your laptop's native Python installation first just to see what it looks like. Run `sudo pip install -r requirements.txt` and then run `python app.py`.

You should then be able to open a web browser, go to http://localhost:5000, and see the message "Hello! I am a Flask application".

This is totally optional - but some people like to see what the app's supposed to do before they try to Dockerize it.

1.4. **Dockerfile**

A **Dockerfile** is a text file that contains a list of commands that the Docker daemon calls while creating an image. The Dockerfile contains all the information that Docker needs to know to run the app — a base Docker image to run from, location of your project code, any dependencies it has, and what commands to run at start-up. It is a simple way to automate the image creation process. The best part is that the commands you write in a Dockerfile are almost identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own Dockerfiles.

We want to create a Docker image with this web app. As mentioned above, all user images are based on a base image. Since our application is written in Python, we will build our own Python image based on `Alpine`. We'll do that using a Dockerfile.

Create a file called Dockerfile in the `flask` directory, and add content to it as described below. Since you are currently in `templates` directory, you need to go up one directory up before you can create your Dockerfile

```
cd ..
```

```
# our base image
FROM alpine:3.5

# install python and pip
RUN apk add --update py2-pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

Now let's see what each of those lines mean..

1.4.1 We'll start by specifying our base image, using the FROM keyword:

```
FROM alpine:3.5
```

1.4.2. The next step usually is to write the commands of copying the files and installing the dependencies. But first we will install the Python pip package to the alpine linux distribution. This will not just install the pip package but any other dependencies too, which includes the python interpreter. Add the following `RUN` command next:

```
RUN apk add --update py2-pip
```

1.4.3. Let's add the files that make up the Flask Application. Install all Python requirements for our app to run. This will be accomplished by adding the lines:

```
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
```

1.4.4. Copy the files you have created earlier into our image by using `COPY` command.

```
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/
```

1.4.5. Specify the port number which needs to be exposed. Since our flask app is running on 5000 that's what we'll expose.

```
EXPOSE 5000
```

1.4.6. The last step is the command for running the application which is simply - `python ./app.py`. Use the `CMD` command to do that:

```
CMD ["python", "/usr/src/app/app.py"]
```

The primary purpose of `CMD` is to tell the container which command it should run by default when it is started.

2. Build the image

Now that you have your Dockerfile, you can build your image. The `docker build` command does the heavy-lifting of creating a docker image from a Dockerfile.

The `docker build command` is quite simple - it takes an optional tag name with the `-t` flag, and the location of the directory containing the Dockerfile - the `.` indicates the current directory:

Note: When you run the `docker build` command given below, make sure to replace `<YOUR_DOCKERHUB_USERNAME>` with your username. This username should be the same one you created when registering on Docker hub. If you haven't done that yet, please go ahead and create an account in Dockerhub.

```
YOUR_DOCKERHUB_USERNAME=<YOUR_DOCKERHUB_USERNAME>
```

For example this is how I assign my dockerhub username

```
YOUR_DOCKERHUB_USERNAME=upendradevisetty
```

Now build the image using the following command:

```
$ docker build -t $YOUR_DOCKERHUB_USERNAME/myfirstapp .
Sending build context to Docker daemon   7.68kB
Step 1/8 : FROM alpine:3.5
 ---> 88e169ea8f46
Step 2/8 : RUN apk add --update py2-pip
 ---> Using cache
 ---> 8b1f026c3899
Step 3/8 : COPY requirements.txt /usr/src/app/
 ---> Using cache
 ---> 6923f451ee09
Step 4/8 : RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
 ---> Running in fb6b7b8beb3c
Collecting Flask==0.10.1 (from -r /usr/src/app/requirements.txt (line 1))
  Downloading Flask-0.10.1.tar.gz (544kB)
Collecting Werkzeug>=0.7 (from Flask==0.10.1->-r /usr/src/app/requirements.txt (line␣
↪1))
  Downloading Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
Collecting Jinja2>=2.4 (from Flask==0.10.1->-r /usr/src/app/requirements.txt (line 1))
  Downloading Jinja2-2.10-py2.py3-none-any.whl (126kB)
Collecting itsdangerous>=0.21 (from Flask==0.10.1->-r /usr/src/app/requirements.txt␣
↪(line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask==0.10.1->-r /usr/src/app/
↪requirements.txt (line 1))
  Downloading MarkupSafe-1.0.tar.gz
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, itsdangerous, Flask
  Running setup.py install for MarkupSafe: started
    Running setup.py install for MarkupSafe: finished with status 'done'
  Running setup.py install for itsdangerous: started
    Running setup.py install for itsdangerous: finished with status 'done'
  Running setup.py install for Flask: started
    Running setup.py install for Flask: finished with status 'done'
Successfully installed Flask-0.10.1 Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1␣
↪itsdangerous-0.24
You are using pip version 9.0.0, however version 9.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
 ---> 16d47a8073fd
Removing intermediate container fb6b7b8beb3c
Step 5/8 : COPY app.py /usr/src/app/
 ---> 338019e5711f
Step 6/8 : COPY templates/index.html /usr/src/app/templates/
```

(continues on next page)

(continued from previous page)

```
 ---> b65ed769c446
Step 7/8 : EXPOSE 5000
 ---> Running in b95001d36e4d
 ---> 0deaa29ca54a
Removing intermediate container b95001d36e4d
Step 8/8 : CMD python /usr/src/app/app.py
 ---> Running in 4a8e82f87e2f
 ---> 40a121fff878
Removing intermediate container 4a8e82f87e2f
Successfully built 40a121fff878
Successfully tagged upendradevisetty/myfirstapp:latest
```

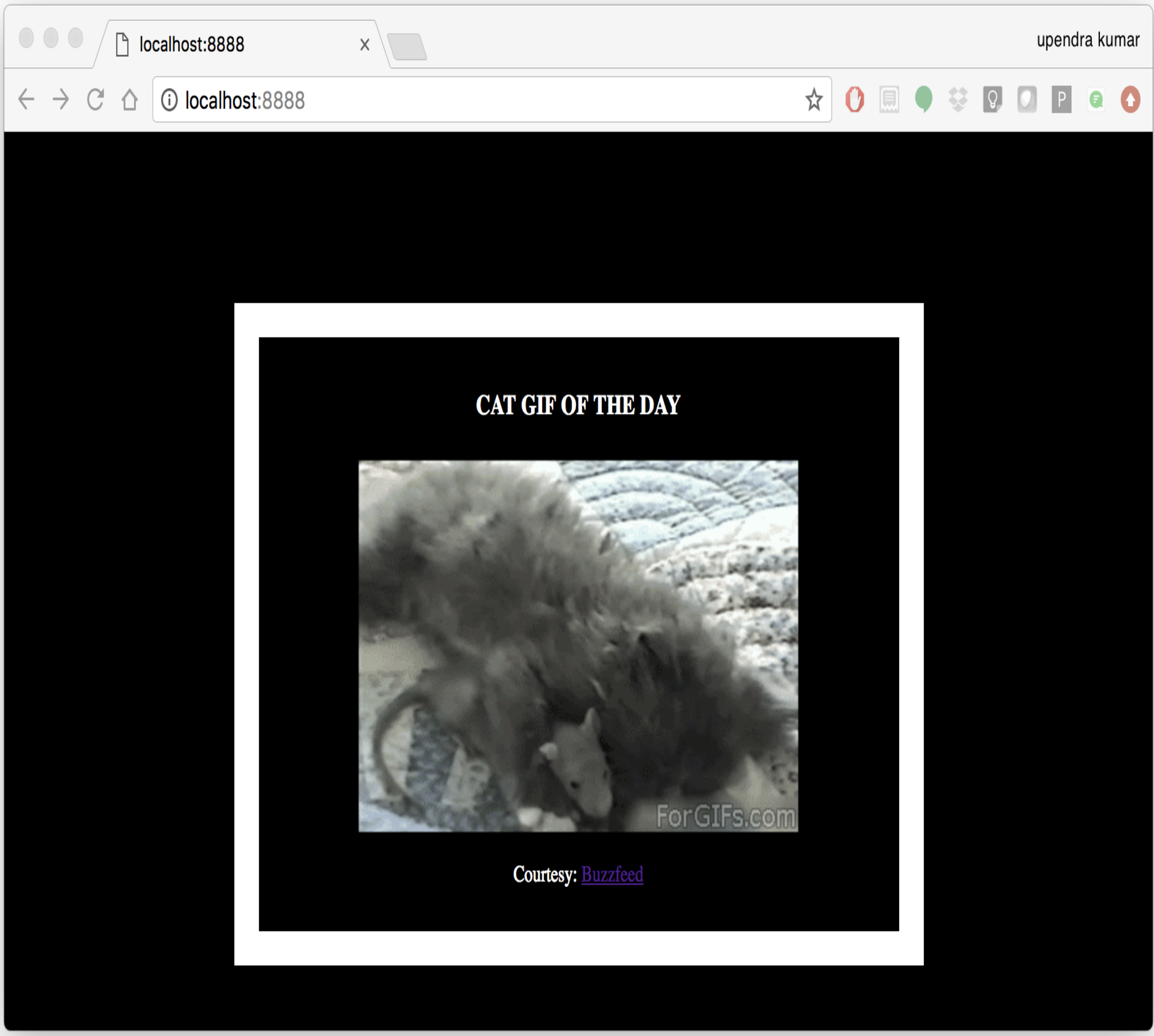
If you don't have the `alpine:3.5 image`, the client will first pull the image and then create your image. Therefore, your output on running the command will look different from mine. If everything went well, your image should be ready! Run `docker images` and see if your image `$YOUR_DOCKERHUB_USERNAME/myfirstapp` shows.

3. Run your image

When Docker can successfully build your Dockerfile, test it by starting a new container from your new image using the docker run command. Don't forget to include the port forwarding options you learned about before.

```
$ docker run -d -p 8888:5000 --name myfirstapp $YOUR_DOCKERHUB_USERNAME/myfirstapp
```

Head over to `http://localhost:8888` and your app should be live.



Hit the Refresh button in the web browser to see a few more cat images.

10.4.3 Exercise (5-10 mins): Deploy a custom Docker image

- Download the sample code from https://github.com/Azure-Samples/docker-django-webapp-linux.git
- Build the image using the Dockerfile in that repo using `docker build` command
- Run an instance from that image

- Verify the web app and container are functioning correctly
- Share your (non-localhost) url on Slack

10.5 5. Dockerfile commands summary

Here's a quick summary of the few basic commands we used in our Dockerfile.

- **FROM** starts the Dockerfile. It is a requirement that the Dockerfile must start with the FROM command. Images are created in layers, which means you can use another image as the base image for your own. The FROM command defines your base layer. As arguments, it takes the name of the image. Optionally, you can add the Dockerhub username of the maintainer and image version, in the format username/imagename:version.
- **RUN** is used to build up the Image you're creating. For each RUN command, Docker will run the command then create a new layer of the image. This way you can roll back your image to previous states easily. The syntax for a RUN instruction is to place the full text of the shell command after the RUN (e.g., RUN mkdir /user/local/foo). This will automatically run in a /bin/sh shell. You can define a different shell like this: RUN /bin/bash -c 'mkdir /user/local/foo'
- **COPY** copies local files into the container.
- **CMD** defines the commands that will run on the Image at start-up. Unlike a RUN, this does not create a new layer for the Image, but simply runs the command. There can only be one CMD per a Dockerfile/Image. If you need to run multiple commands, the best way to do that is to have the CMD run a script. CMD requires that you tell it where to run the command, unlike RUN. So example CMD commands would be:

```
CMD ["python", "./app.py"]

CMD ["/bin/bash", "echo", "Hello World"]
```

- EXPOSE creates a hint for users of an image which ports provide services. It is included in the information which can be retrieved via `$ docker inspect <container-id>`.

Note: The EXPOSE command does not actually make any ports accessible to the host! Instead, this requires publishing ports by means of the `-p` flag when using `docker run`.

- PUSH pushes your image to Docker Cloud, or alternately to a private registry

Note: If you want to learn more about Dockerfiles, check out Best practices for writing Dockerfiles.

10.6 6. Demo's

10.6.1 6.1 Portainer

Portainer is an open-source lightweight managment UI which allows you to easily manage your Docker hosts or Swarm cluster.

- Simple to use: It has never been so easy to manage Docker. Portainer provides a detailed overview of Docker and allows you to manage containers, images, networks and volumes. It is also really easy to deploy, you are just one Docker command away from running Portainer anywhere.

- Made for Docker: Portainer is meant to be plugged on top of the Docker API. It has support for the latest versions of Docker, Docker Swarm and Swarm mode.

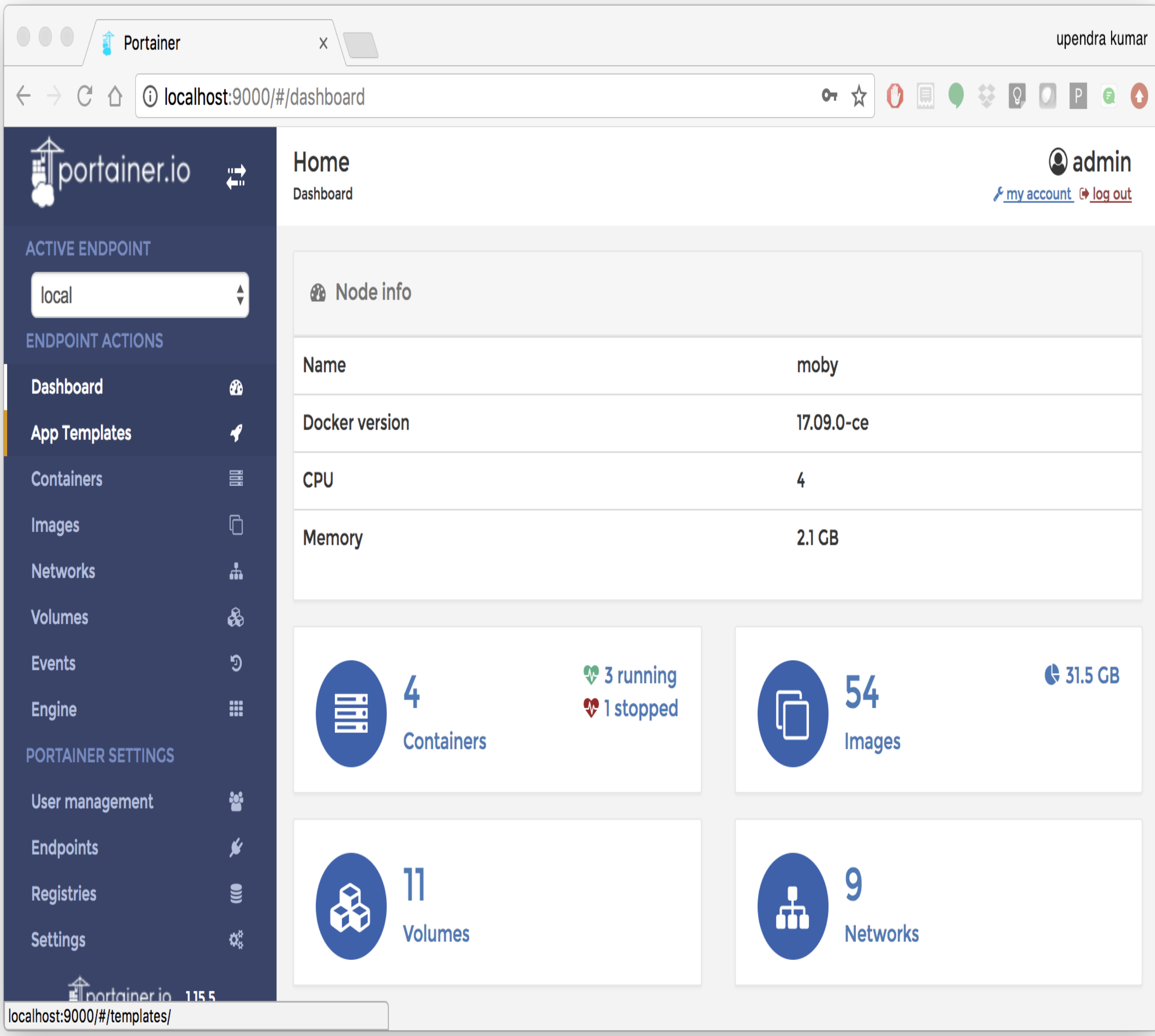
6.1.1 Installation

Use the following Docker commands to deploy Portainer. Now the second line of command should be familiar to you by now. We will talk about first line of command in the Advanced Docker session.

```
$ docker volume create portainer_data

$ docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock -v␣
↪portainer_data:/data portainer/portainer
```

- If you are on mac, you'll just need to access the port 9000 (http://localhost:9000) of the Docker engine where portainer is running using username `admin` and password `tryportainer`
- If you are running Docker on Atmosphere/Jetstream or on any other cloud, you can open `ipaddress:9000`. For my case this is `http://128.196.142.26:9000`

Note: The *-v /var/run/docker.sock:/var/run/docker.sock* option can be used in mac/linux environments only.



Name	moby
Docker version	17.09.0-ce
CPU	4
Memory	2.1 GB

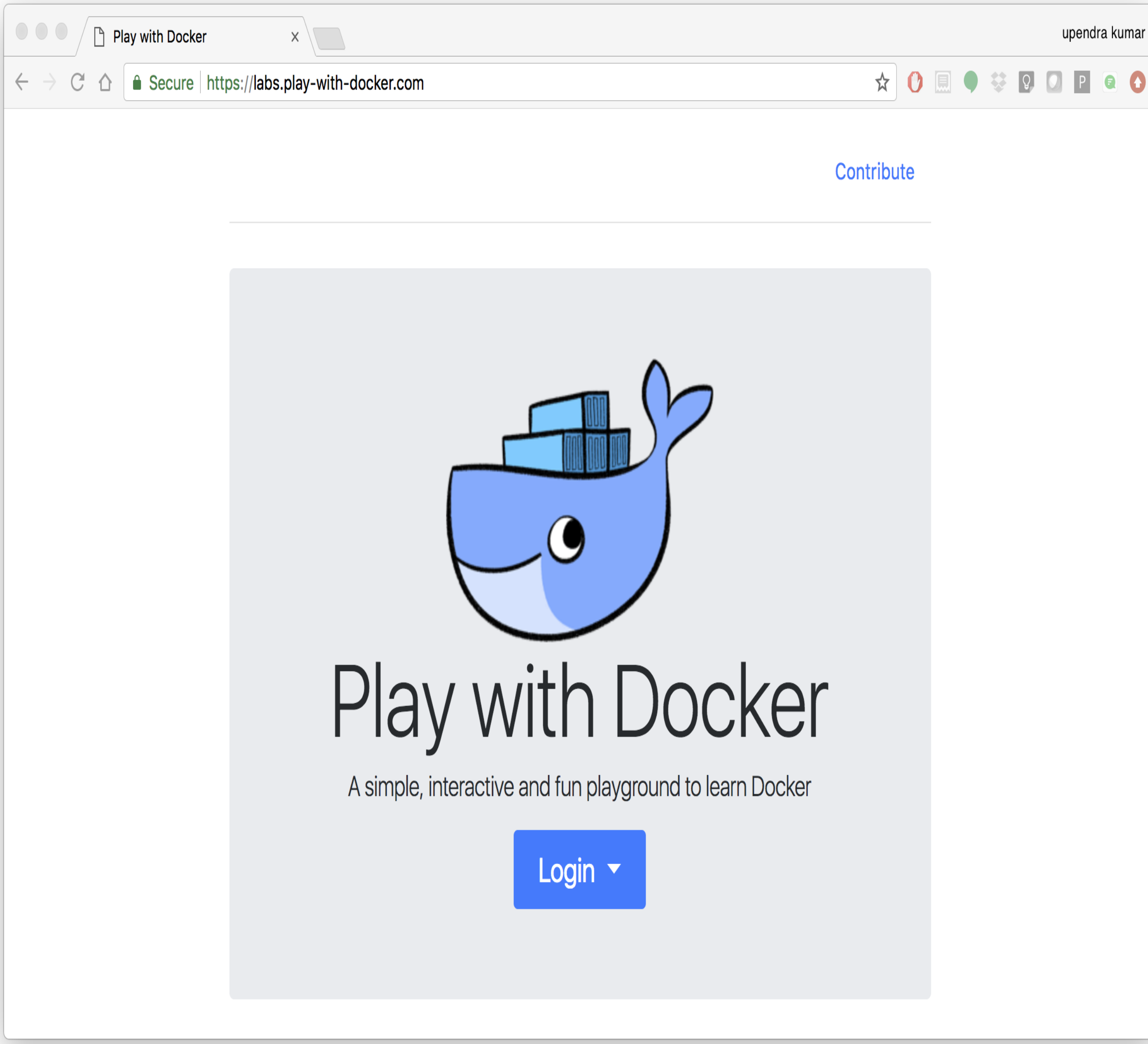
10.6.2 6.2 Play-with-docker (PWD)

PWD is a Docker playground which allows users to run Docker commands in a matter of seconds. It gives the experience of having a free Alpine Linux Virtual Machine in browser, where you can build and run Docker containers and even create clusters in Docker Swarm Mode. Under the hood, Docker-in-Docker (DinD) is used to give the effect of multiple VMs/PCs. In addition to the playground, PWD also includes a training site composed of a large set of Docker labs and quizzes from beginner to advanced level available at training.play-with-docker.com.

6.2.1 Installation

You don't have to install anything to use PWD. Just open `https://labs.play-with-docker.com/` and start using PWD

Note: You can use your Dockerhub credentials to log-in to PWD


Play with Docker
upendra kumar
Secure
https://labs.play-with-docker.com
Contribute
Play with Docker
A simple, interactive and fun playground to learn Docker
Login

CHAPTER 11

Advanced Docker

Now that we are relatively comfortable with Docker basics, lets look at some of the advanced Docker topics such as porting the Docker image to repositories (public and private), managing data in containers and finally deploy containers into cloud and other infrastructures etc.,

11.1 1. Docker registries

To demonstrate the portability of what we just created, let's upload our built Docker image and run it somewhere else (Atmosphere cloud). After all, you'll need to learn how to push to registries when you want to deploy containers to production.

Important: So what exactly is a registry?

A registry is a collection of repositories, and a repository is a collection of images—sort of like a GitHub repository, except the code is already built. An account on a registry can create many repositories. The docker CLI uses Docker's public registry by default. You can even set up your own private registry using Docker Trusted Registry

There are several things you can do with Docker registries:

- Pushing images
- Finding images
- Pulling images
- Sharing images

11.1.1 1.1 Public repositories

Some example of public registries include Docker cloud, Docker hub and quay.io.

1.1.1 Log in with your Docker ID

Now that you've created and tested your image, you can push it to Docker cloud or Docker hub.

Note: If you don't have a Docker account, sign up for one at Docker cloud or Docker hub. Make note of your username. There are several advantages of registering to Dockerhub which we will see later on in the session

First you have to login to your Docker hub account. To do that:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have␣
↪a Docker ID, head over to https://hub.docker.com to create one.
Username (upendradevisetty):
Password:
```

Enter Username and Password when prompted.

1.1.2 Tag the image

The notation for associating a local image with a repository on a registry is `username/repository:tag`. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as `get-started:part2`. This will put the image in the `get-started` repository and tag it as `part2`.

Note: By default the docker image gets a `latest` tag if you don't provide one. Thought convenient, it is not recommended for reproducibility purposes.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image will upload to your desired destination. For our docker image since we already have our Dockerhub username we will just add tag which in this case is `1.0`

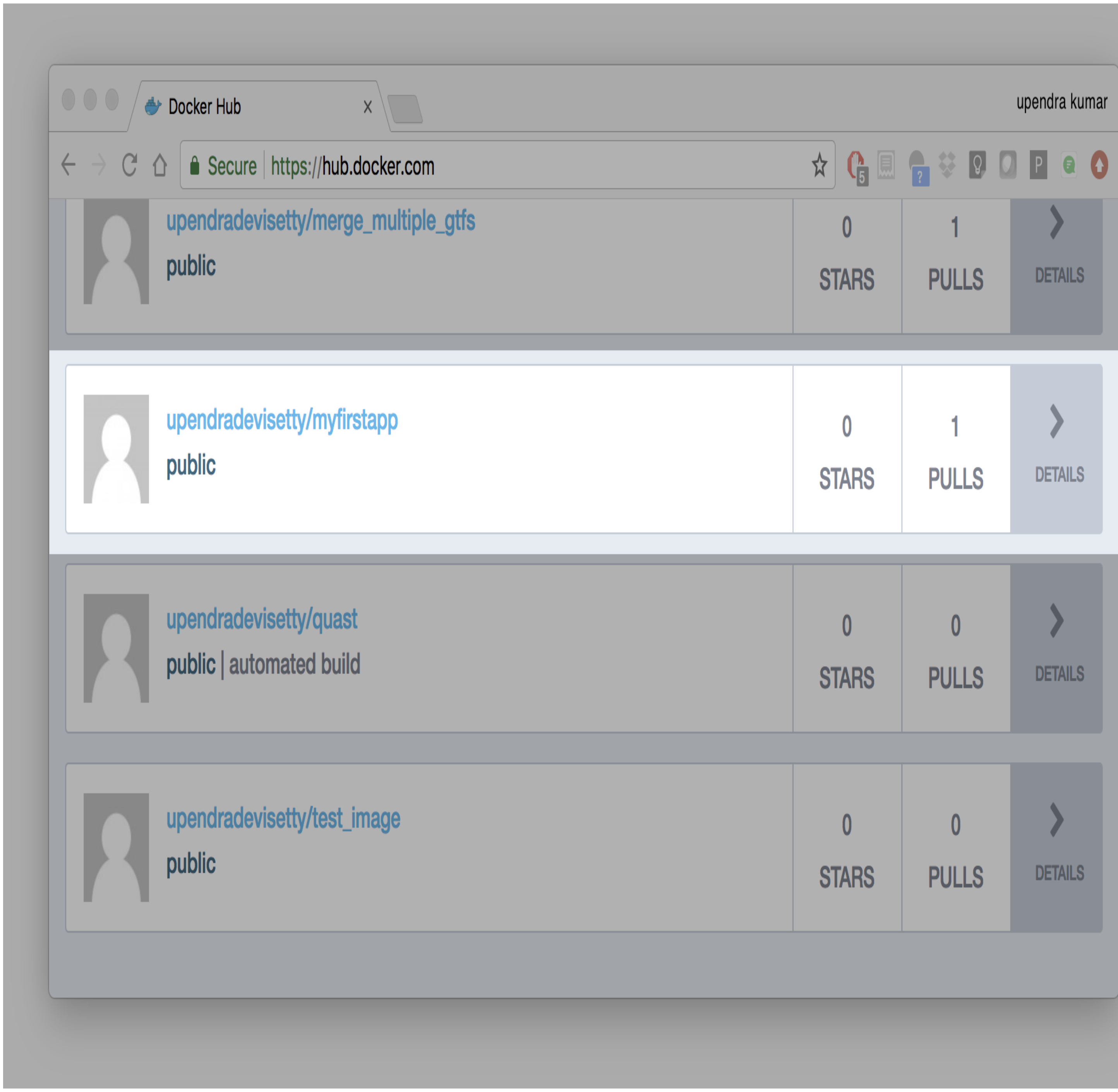
```
$ docker tag $YOUR_DOCKERHUB_USERNAME/myfirstapp $YOUR_DOCKERHUB_USERNAME/
↪myfirstapp:1.0
```

1.1.3 Publish the image

Upload your tagged image to the Dockerhub repository

```
$ docker push $YOUR_DOCKERHUB_USERNAME/myfirstapp:1.0
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you will see the new image there, with its pull command.



Congrats! You just made your first Docker image and shared it with the world!

1.1.4 Pull and run the image from the remote repository

Let's try to run the image from the remote repository on Cloud server by logging into CyVerse Atmosphere, launching an instance

First install Docker on Atmosphere using from here `https://docs.docker.com/install/linux/docker-ce/ubuntu` or alternatively you can use `ezd` command which is a short-cut command for installing

Docker on Atmosphere

```
$ ezd
```

Now run the following command to run the docker image from Dockerhub

```
$ sudo docker run -d -p 8888:5000 --name myfirstapp $YOUR_DOCKERHUB_USERNAME/
↪myfirstapp:1.0
```

Note: You don't have to run `docker pull` since if the image isn't available locally on the machine, Docker will pull it from the repository.

Head over to `http://<ipaddress>:8888` and your app should be live.

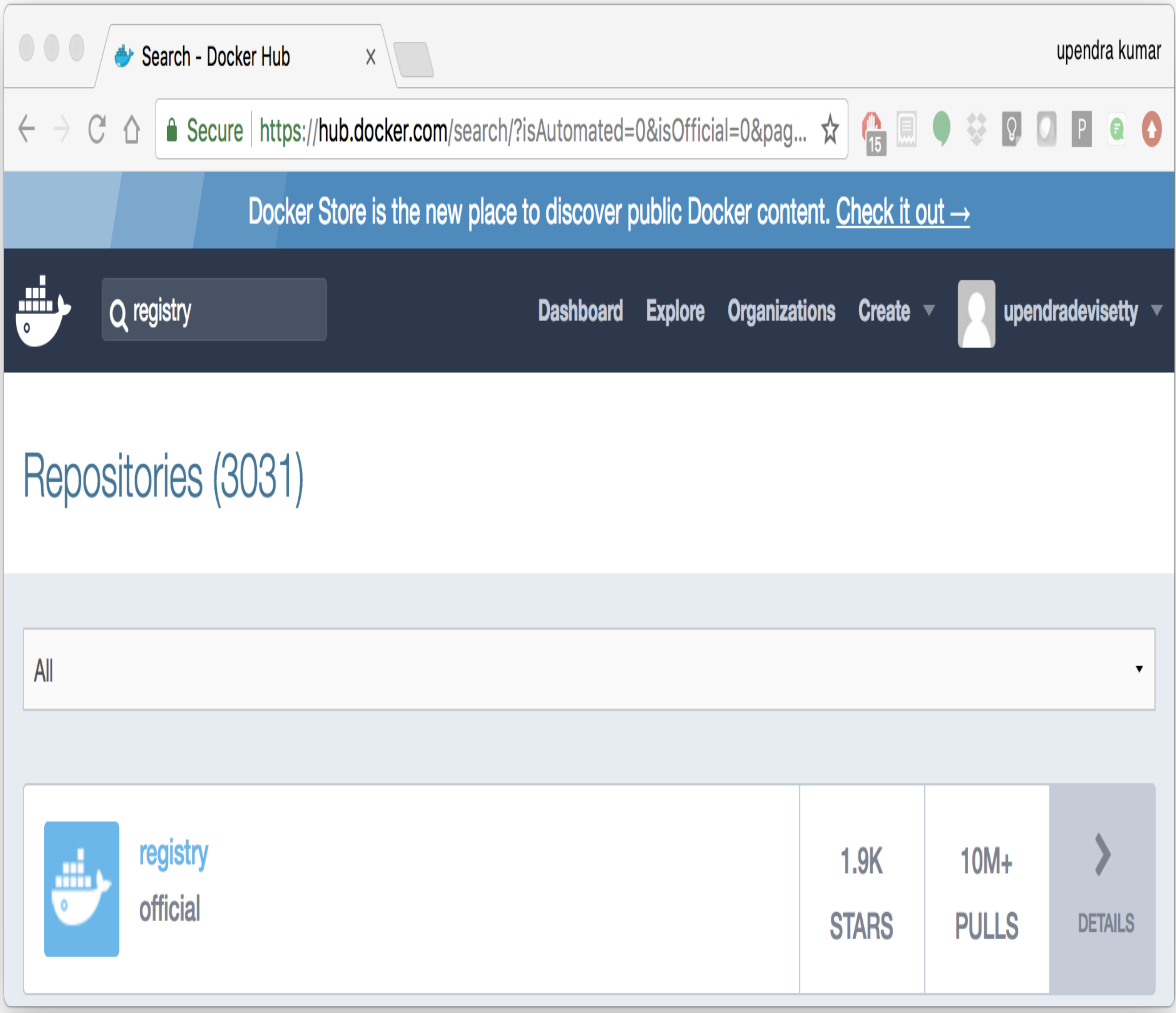
11.1.2 1.2 Private repositories

In an earlier part, we had looked at the Docker Hub, which is a public registry that is hosted by Docker. While the Dockerhub plays an important role in giving public visibility to your Docker images and for you to utilize quality Docker images put up by others, there is a clear need to setup your own private registry too for your team/organization. For example, CyVerse has it own private registry which will be used to push the Docker images.

1.2.1 Pull down the Registry Image

You might have guessed by now that the registry must be available as a Docker image from the Docker Hub and it should be as simple as pulling the image down and running that. You are correct!

A Dockerhub search on the keyword `registry` brings up the following image as the top result:



Run a container from `registry` Dockerhub image

```
$ docker run -d -p 5000:5000 --name registry registry:2
```

Run `docker ps -l` to check the recent container from this Docker image

```
$ docker ps -l
CONTAINER ID        IMAGE               COMMAND                  CREATED             ␣
↪STATUS              PORTS                    NAMES
```

(continues on next page)

(continued from previous page)

```
6e44a0459373        registry:2          "/entrypoint.sh /e..."   11 seconds ago     ␣
↪Up 10 seconds        0.0.0.0:5000->5000/tcp   registry
```

1.2.2 Tag the image that you want to push

Next step is to tag your image under the registry namespace and push it there

```
$ REGISTRY=localhost:5000

$ docker tag $YOUR_DOCKERHUB_USERNAME/myfirstapp:1.0 $REGISTRY/$(whoami)/myfirstapp:1.
↪0
```

1.2.2 Publish the image into the local registry

Finally push the image to the local registry

```
$ docker push $REGISTRY/$(whoami)/myfirstapp:1.0
The push refers to a repository [localhost:5000/upendra_35/myfirstapp]
64436820c85c: Pushed
831cff83ec9e: Pushed
c3497b2669a8: Pushed
1c5b16094682: Pushed
c52044a91867: Pushed
60ab55d3379d: Pushed
1.0: digest: sha256:5095dea8b2cf308c5866ef646a0e84d494a00ff0e9b2c8e8313a176424a230ce␣
↪size: 1572
```

1.2.3 Pull and run the image from the local repository

You can also pull the image from the local repository similar to how you pull it from Dockerhub and run a container from it

```
$ docker run -d -P --name=myfirstapplocal $REGISTRY/$(whoami)/myfirstapp:1.0
```

11.2 2. Automated Docker image building from github

An automated build is a Docker image build that is triggered by a code change in a GitHub or Bitbucket repository. By linking a remote code repository to a Dockerhub automated build repository, you can build a new Docker image every time a code change is pushed to your code repository.

A build context is a Dockerfile and any files at a specific location. For an automated build, the build context is a repository containing a Dockerfile.

Automated Builds have several advantages:

- Images built in this way are built exactly as specified.
- The Dockerfile is available to anyone with access to your Docker Hub repository.
- Your repository is kept up-to-date with code changes automatically.
- Automated Builds are supported for both public and private repositories on both GitHub and Bitbucket.

11.2.1 2.1 Prerequisites

To use automated builds, you first must have an account on Docker Hub and on the hosted repository provider (GitHub or Bitbucket). While Dockerhub supports linking both GitHub and Bitbucket repositories, here we will use a GitHub repository. If you don't already have one, make sure you have a GitHub account. A basic account is free

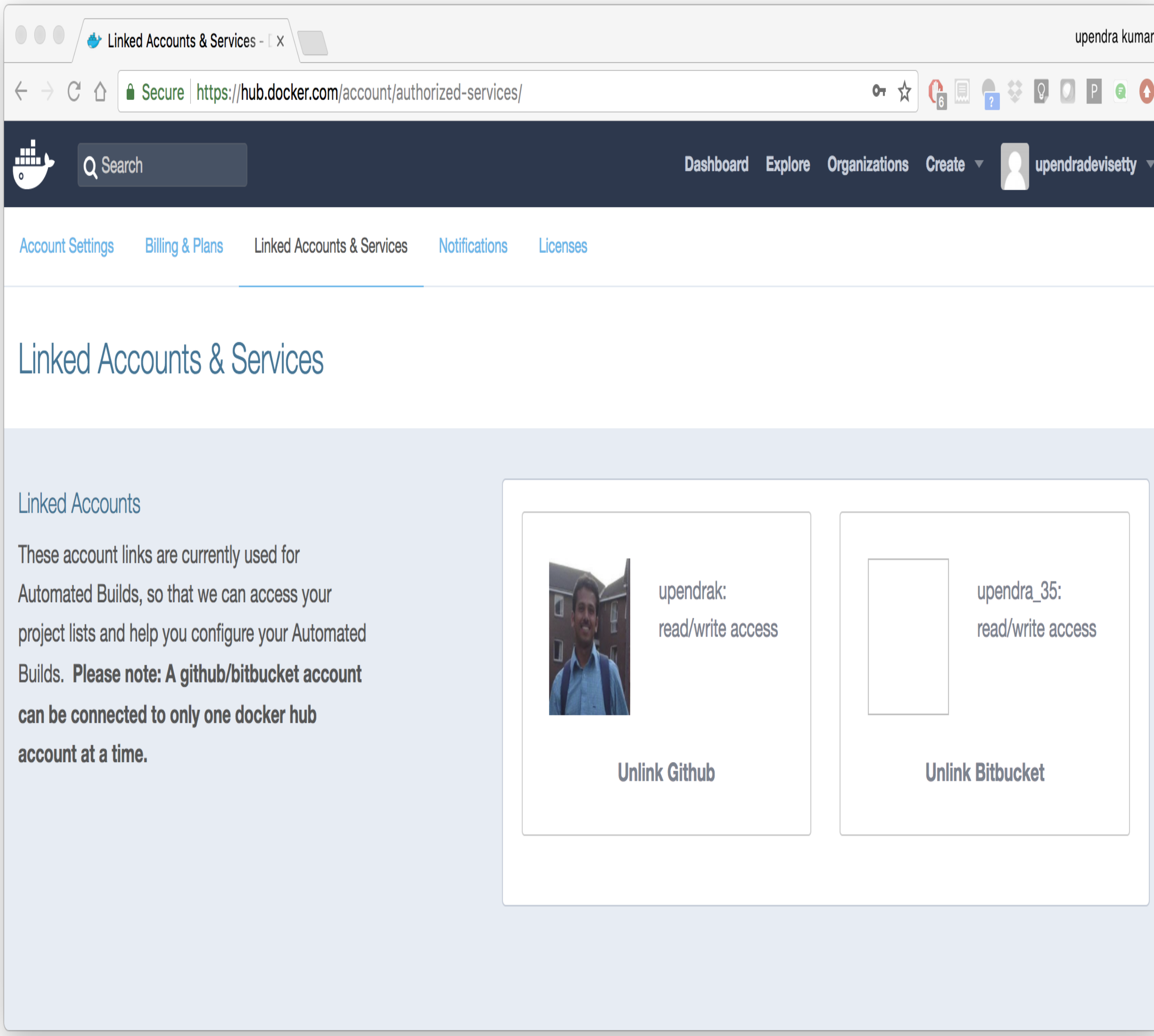
Note:

- If you have previously linked your Github or Bitbucket account, you must have chosen the Public and Private connection type. To view your current connection settings, log in to Docker Hub and choose Profile > Settings > Linked Accounts & Services.
- Building Windows containers is not supported.

11.2.2 2.2 Link your Docker Hub account to GitHub

1. Log into Docker Hub.
2. Navigate to Profile > Settings > Linked Accounts & Services.
3. Click the `Link GitHub`. The system prompts you to choose between **Public and Private** and **Limited Access**. The **Public** and **Private** connection type is required if you want to use the Automated Builds.
4. Press `Select` under **Public and Private** connection type. If you are not logged into GitHub, the system prompts you to enter GitHub credentials before prompting you to grant access. After you grant access to your code repository, the system returns you to Docker Hub and the link is complete.

After you grant access to your code repository, the system returns you to Docker Hub and the link is complete. For example, github linked hosted repository looks like this:



Linked Accounts & Services

Linked Accounts

These account links are currently used for Automated Builds, so that we can access your project lists and help you configure your Automated Builds. **Please note: A github/bitbucket account can be connected to only one docker hub account at a time.**

11.2.3 2.3 Create a new automated build

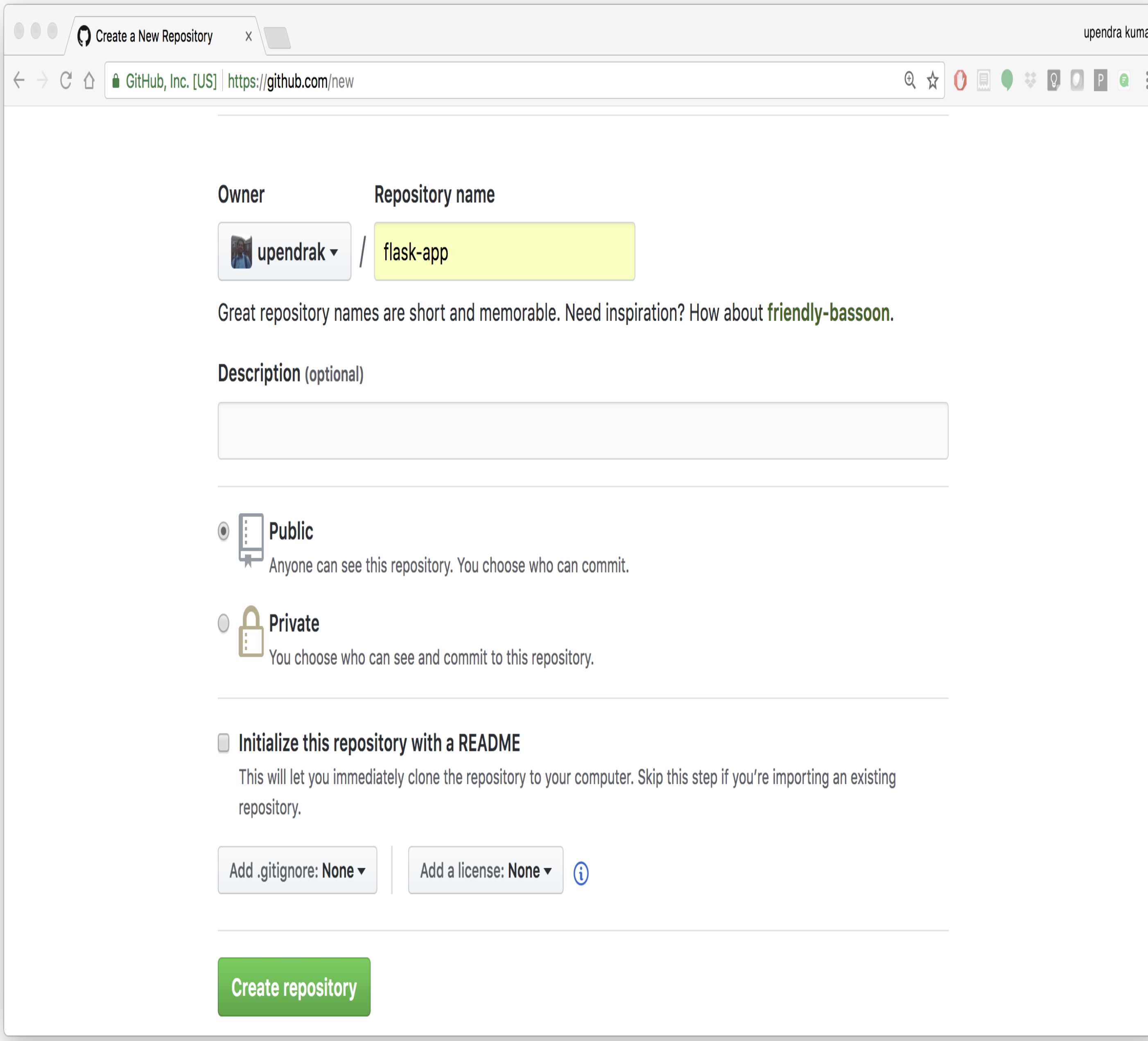
Automated build repositories rely on the integration with your github code repository to build.

Let's create an automatic build for our `flask-app` using the instructions below:

1. Initialize git repository for the *flask-app* directory

```
$ git init
Initialized empty Git repository in /Users/upendra_35/Documents/git.repos/flask-app/.
↪git/

$ git status
On branch master

Initial commit

Untracked files:
(use "git add <file>..." to include in what will be committed)

        Dockerfile
        app.py
        requirements.txt
        templates/

nothing added to commit but untracked files present (use "git add" to track)

$ git add * && git commit -m"Add files and folders"
[master (root-commit) cfdf021] Add files and folders
 4 files changed, 75 insertions(+)
 create mode 100644 Dockerfile
 create mode 100644 app.py
 create mode 100644 requirements.txt
 create mode 100644 templates/index.html
```

2. Create a new repository on github by navigating to this url - https://github.com/new


Create a New Repository

upendra kuma

GitHub, Inc. [US] https://github.com/new

Owner

upendrak

Repository name

flask-app

Great repository names are short and memorable. Need inspiration? How about friendly-bassoon.

Description (optional)

Public
Anyone can see this repository. You choose who can commit.

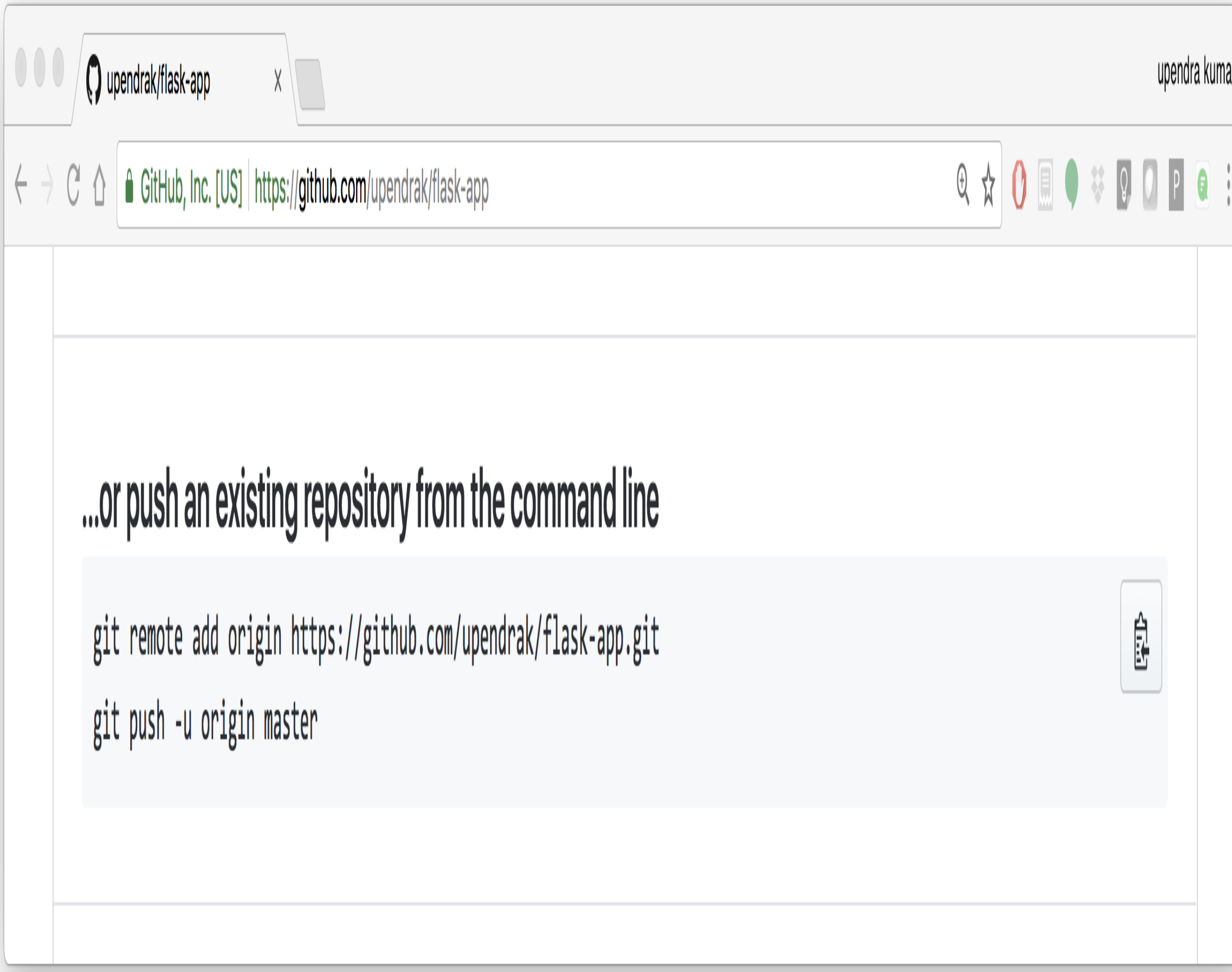
Private
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

3. Push the repository to github



...or push an existing repository from the command line

```
git remote add origin https://github.com/upendrak/flask-app.git
git push -u origin master
```

```
$ git remote add origin https://github.com/upendrak/flask-app.git

$ git push -u origin master
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 1.44 KiB | 0 bytes/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/upendrak/flask-app.git
```

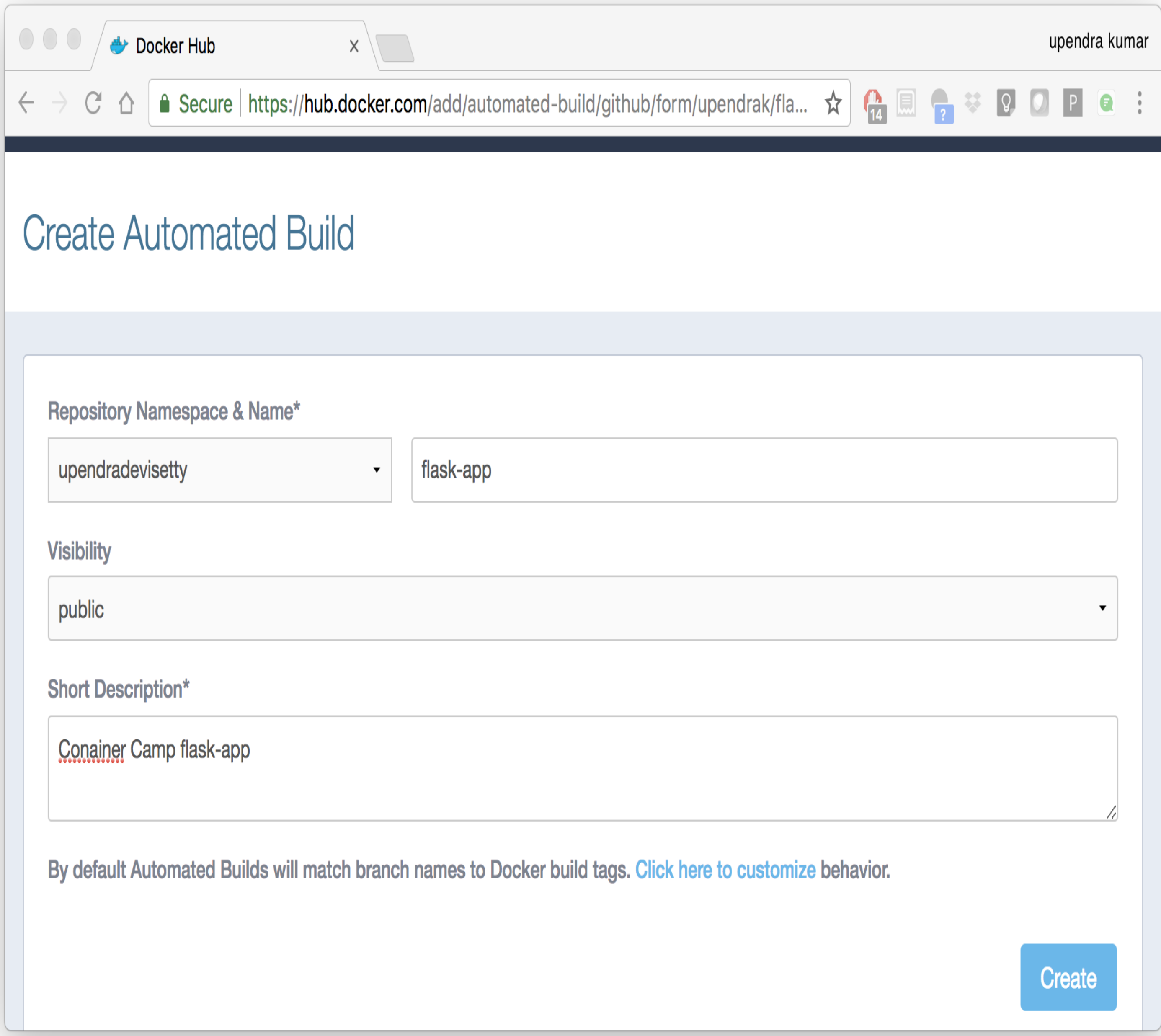
(continues on next page)

(continued from previous page)

```
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

4. Select `Create` > `Create Automated Build` from Docker Hub.

- The system prompts you with a list of User/Organizations and code repositories.
- For now select your GitHub account from the User/Organizations list on the left. The list of repositories change.
- Pick the project to build. In this case `flask-app`. Type in "Conainer Camp flask-app" in the Short Description box.
- If you have a long list of repos, use the filter box above the list to restrict the list. After you select the project, the system displays the Create Automated Build dialog.


Docker Hub
upendra kumar
Secure https://hub.docker.com/add/automated-build/github/form/upendrak/fla...
Create

Create Automated Build

Repository Namespace & Name*

upendradevisetty / flask-app

Visibility

public

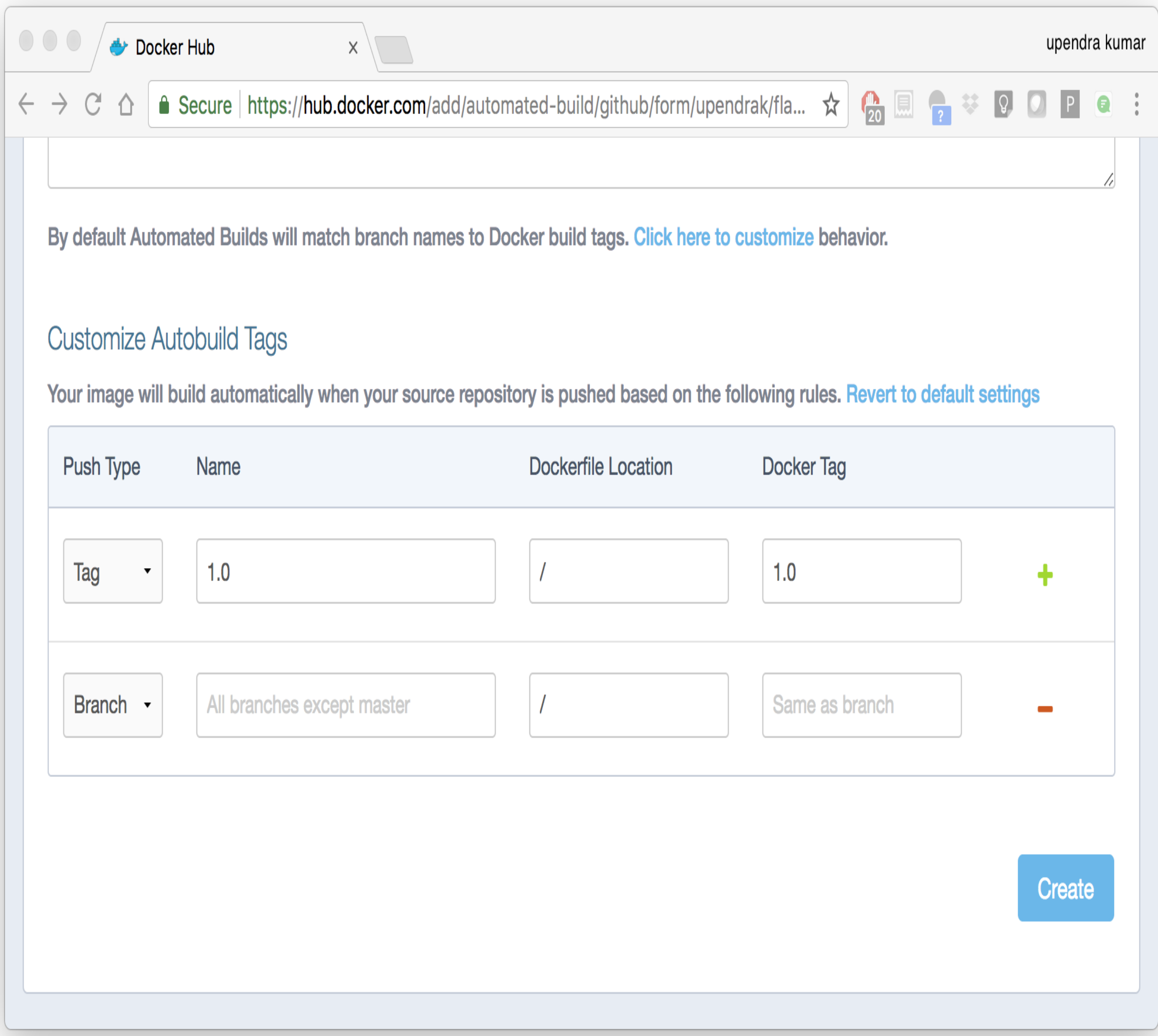
Short Description*

Conainer Camp flask-app

By default Automated Builds will match branch names to Docker build tags. Click here to customize behavior.

Note: The dialog assumes some defaults which you can customize. By default, Docker builds images for each branch in your repository. It assumes the Dockerfile lives at the root of your source. When it builds an image, Docker tags it with the branch name.

5. Customize the automated build by pressing the `Click here to customize` behavior link.


Docker Hub

upendra kumar

Secure https://hub.docker.com/add/automated-build/github/form/upendrak/fla...

By default Automated Builds will match branch names to Docker build tags. Click here to customize behavior.

Customize Autobuild Tags

Your image will build automatically when your source repository is pushed based on the following rules. Revert to default settings

Push Type	Name	Dockerfile Location	Docker Tag	
Tag	1.0	/	1.0	+
Branch	All branches except master	/	Same as branch	−

Create

Specify which code branches or tags to build from. You can build by a code branch or by an image tag. You can enter a specific value or use a regex to select multiple values. To see examples of regex, press the Show More link on the right of the page.

- Enter the `master` (default) for the name of the branch.
- Leave the Dockerfile location as is.
- Recall the file is in the root of your code repository.
- Specify `1.0` for the Tag Name.

6. Click `Create`.

Important: During the build process, Docker copies the contents of your Dockerfile to Docker Hub. The Docker community (for public repositories) or approved team members/orgs (for private repositories) can then view the Dockerfile on your repository page.

The build process looks for a README.md in the same directory as your Dockerfile. If you have a README.md file in your repository, it is used in the repository as the full description. If you change the full description after a build, it's overwritten the next time the Automated Build runs. To make changes, modify the README.md in your Git repository.

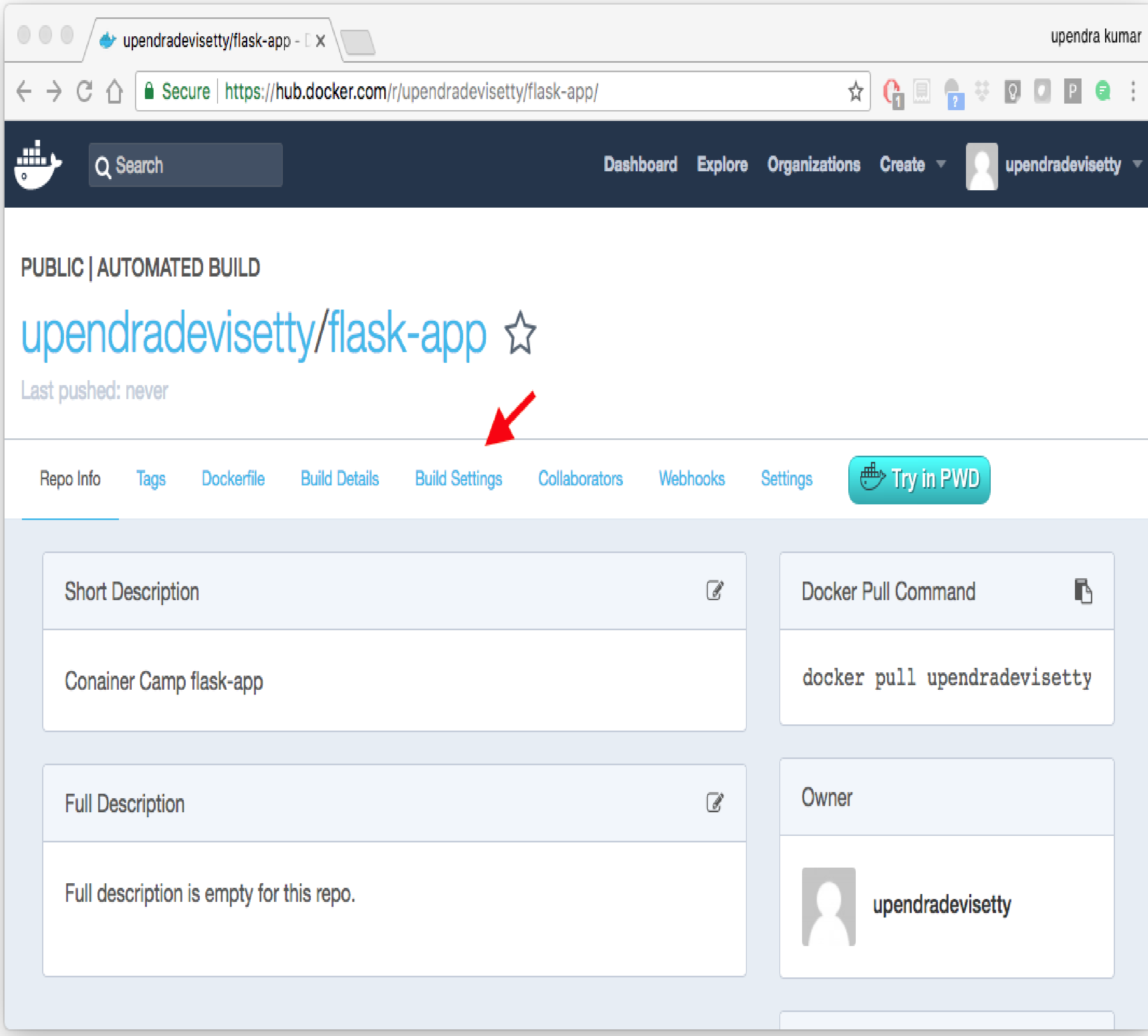
Warning: You can only trigger one build at a time and no more than one every five minutes. If you already have a build pending, or if you recently submitted a build request, Docker ignores new requests.

It can take a few minutes for your automated build job to be created. When the system is finished, it places you in the detail page for your Automated Build repository.

7. Manually Trigger a Build

Before you trigger an automated build by pushing to your GitHub `flask-app` repo, you'll trigger a manual build. Triggering a manual build ensures everything is working correctly.

From your automated build page choose `Build Settings`



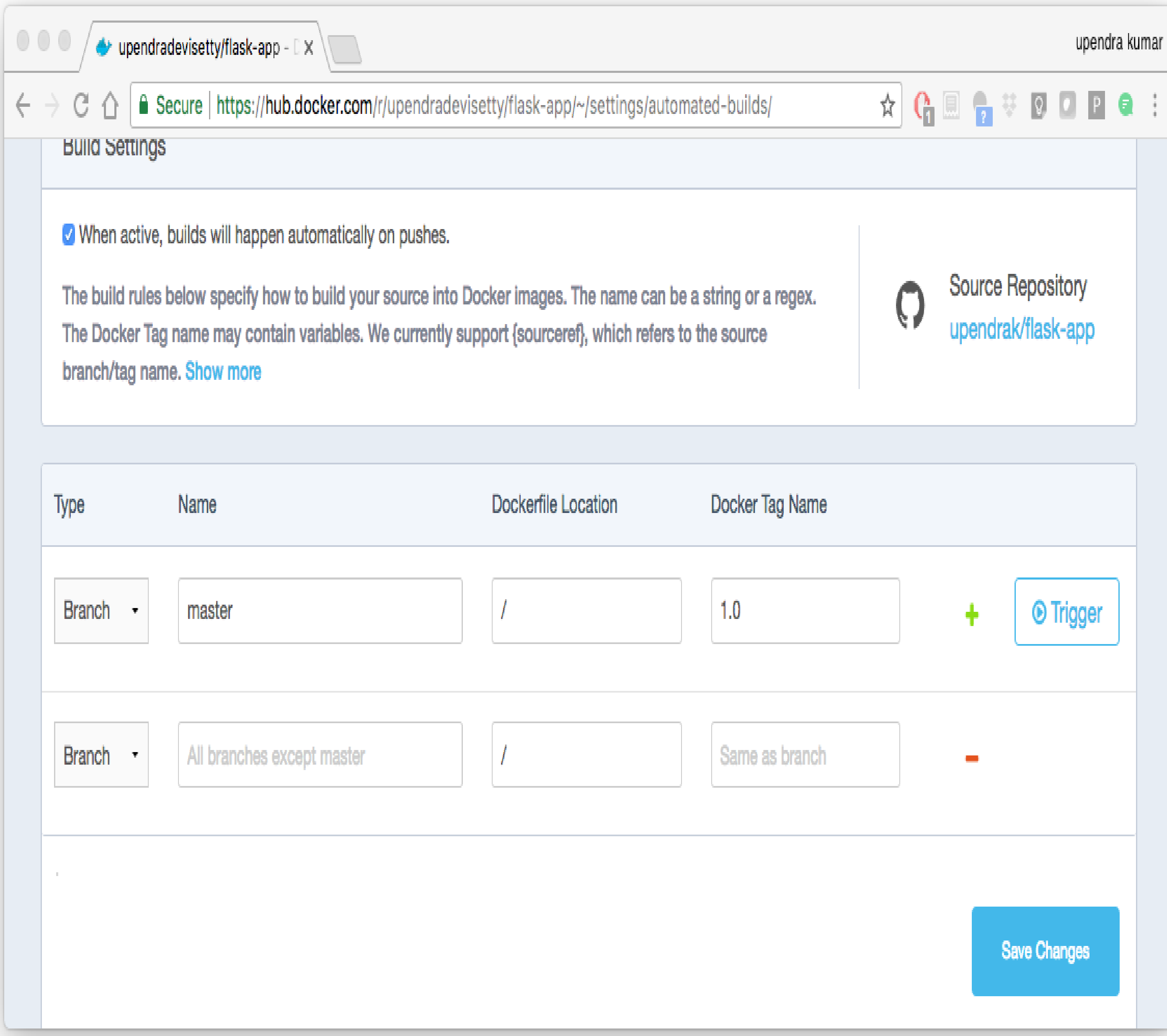
Conainer Camp flask-app

```
docker pull upendradevisetty
```

Full description is empty for this repo.

Press `Trigger` button and finally click `Save Changes`.

Note: Docker builds everything listed whenever a push is made to the code repository. If you specify a particular branch or tag, you can manually build that image by pressing the Trigger. If you use a regular expression syntax (regex) to define your build branch or tag, Docker does not give you the option to manually build.


upendradevisetty/flask-app
upendra kumar
Secure https://hub.docker.com/r/upendradevisetty/flask-app/~/settings/automated-builds/
Build Settings
Source Repository
upendrak/flask-app
Show more
Trigger
Save Changes

When active, builds will happen automatically on pushes.

The build rules below specify how to build your source into Docker images. The name can be a string or a regex. The Docker Tag name may contain variables. We currently support {sourceref}, which refers to the source branch/tag name.

Type	Name	Dockerfile Location	Docker Tag Name
Branch	master	/	1.0
Branch	All branches except master	/	Same as branch

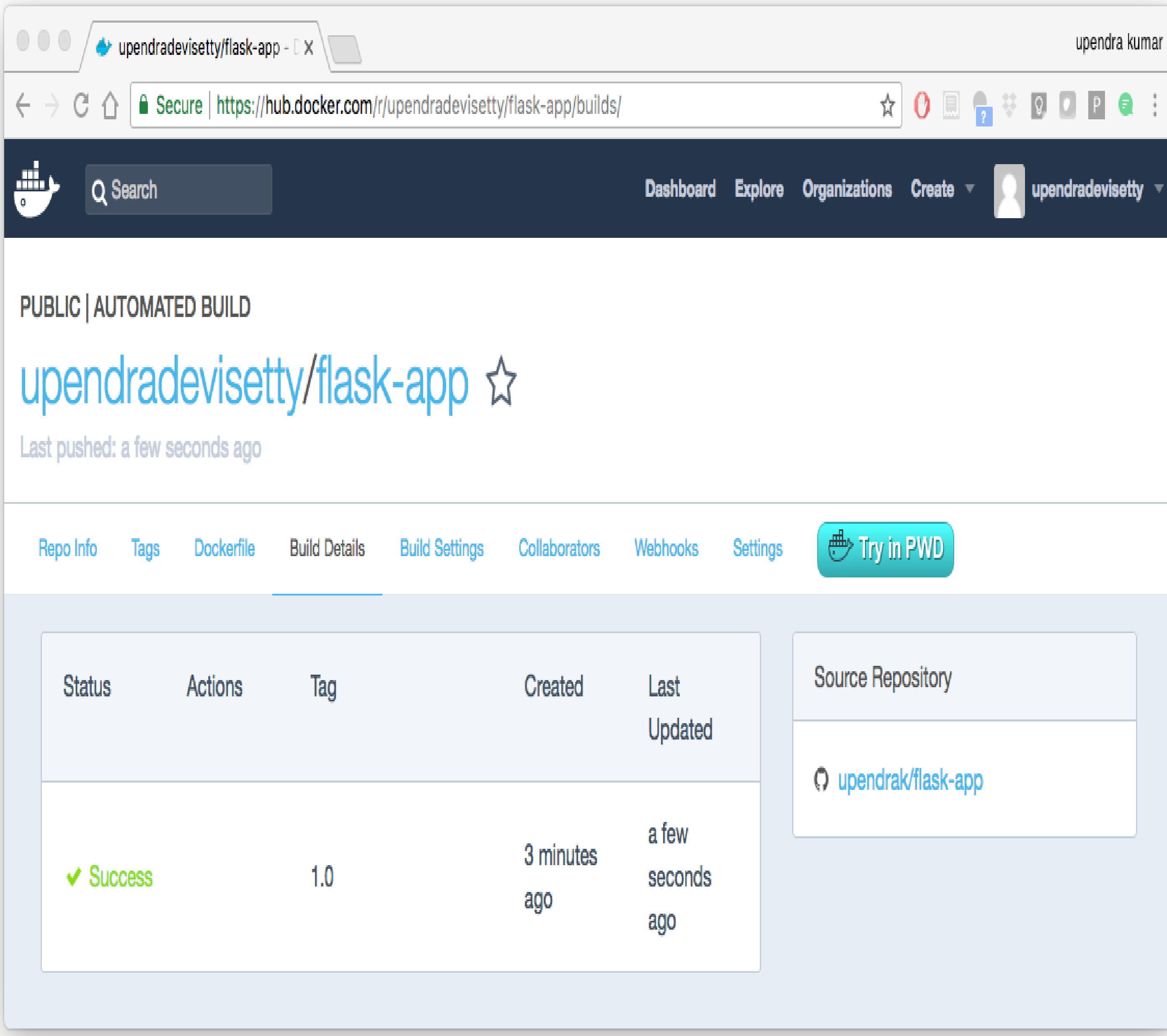
8. Review the build results

The Build Details page shows a log of your build systems:

Navigate to the `Build Details` page.

Wait until your image build is done.

You may have to manually refresh the page and your build may take several minutes to complete.



PUBLIC | AUTOMATED BUILD

upendradevisetty/flask-app

Last pushed: a few seconds ago

Status	Actions	Tag	Created	Last Updated
✓ Success		1.0	3 minutes ago	a few seconds ago

Source Repository
upendrak/flask-app

11.2.4 Exercise 1 (5-10 mins): Updating and automated building

- Add some more cat pics to the *app.py* file
- Add, Commit and Push it to your github repo
- Trigger automatic build with a new tag (2.0) on Dockerhub
- Run an instance to make sure the new pics show up

- Share your Dockerhub link url on Slack

11.3 3. Managing data in Docker

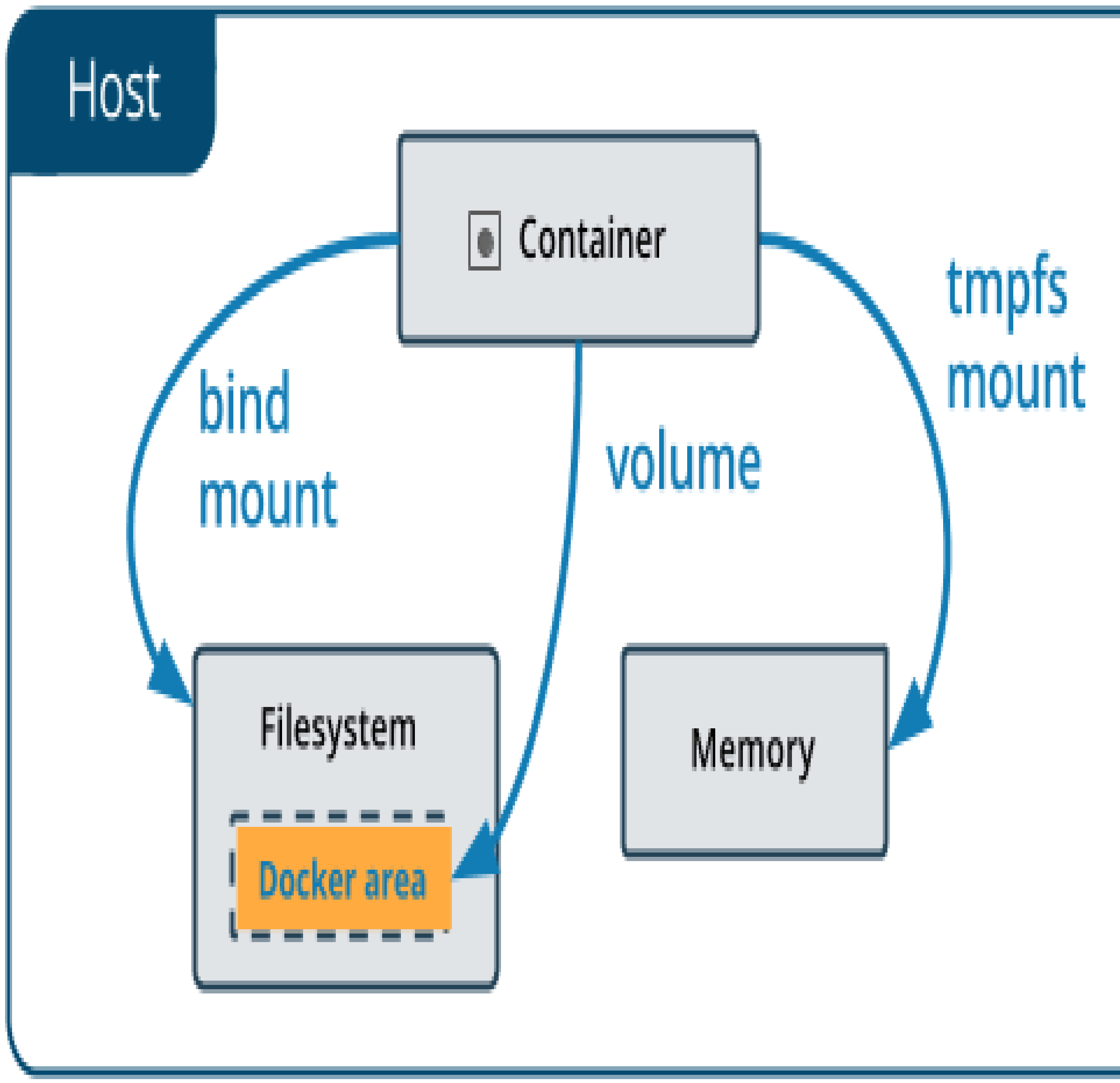
It is possible to store data within the writable layer of a container, but there are some limitations:

- The data doesn't persist when that container is no longer running, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. You can't easily move the data somewhere else.

Docker offers three different ways to mount data into a container from the Docker host: **volumes**, **bind mounts**, or **tmpfs volumes**. When in doubt, volumes are almost always the right choice.

11.3.1 3.1 Volumes

Volumes are created and managed by Docker. You can create a volume explicitly using the `docker volume create` command, or Docker can create a volume during container creation. When you create a volume, it is stored within a directory on the Docker host (`/var/lib/docker/` on Linux and check for the location on mac in here https://timonweb.com/posts/getting-path-and-accessing-persistent-volumes-in-docker-for-mac/). When you mount the volume into a container, this directory is what is mounted into the container. A given volume can be mounted into multiple containers simultaneously. When no running container is using a volume, the volume is still available to Docker and is not removed automatically. You can remove unused volumes using `docker volume prune` command.



Volumes are often a better choice than persisting data in a container's writable layer, because using a volume does not increase the size of containers using it, and the volume's contents exist outside the lifecycle of a given container. While bind mounts (which we will see later) are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.

- Volumes can be more safely shared among multiple containers.
- A new volume's contents can be pre-populated by a container.

Note: If your container generates non-persistent state data, consider using a `tmpfs` mount to avoid storing the data anywhere permanently, and to increase the container's performance by avoiding writing into the container's writable layer.

3.1.1 Choose the -v or –mount flag for mounting volumes

Originally, the `-v` or `--volume` flag was used for standalone containers and the `--mount` flag was used for swarm services. However, starting with Docker 17.06, you can also use `--mount` with standalone containers. In general, `--mount` is more explicit and verbose. The biggest difference is that the `-v` syntax combines all the options together in one field, while the `--mount` syntax separates them. Here is a comparison of the syntax for each flag.

Tip: New users should use the `--mount` syntax. Experienced users may be more familiar with the `-v` or `--volume` syntax, but are encouraged to use `--mount`, because research has shown it to be easier to use.

`-v` or `--volume`: Consists of three fields, separated by colon characters (:). The fields must be in the correct order, and the meaning of each field is not immediately obvious. - In the case of named volumes, the first field is the name of the volume, and is unique on a given host machine. - The second field is the path where the file or directory are mounted in the container. - The third field is optional, and is a comma-separated list of options, such as `ro`.

`--mount`: Consists of multiple key-value pairs, separated by commas and each consisting of a `<key>=<value>` tuple. The `--mount` syntax is more verbose than `-v` or `--volume`, but the order of the keys is not significant, and the value of the flag is easier to understand. - The type of the mount, which can be **bind**, **volume**, or **tmpfs**. - The source of the mount. For named volumes, this is the name of the volume. For anonymous volumes, this field is omitted. May be specified as **source** or **src**. - The destination takes as its value the path where the file or directory is mounted in the container. May be specified as **destination**, **dst**, or **target**. - The readonly option, if present, causes the bind mount to be mounted into the container as read-only.

Note: The `--mount` and `-v` examples have the same end result.

3.1.2. Create and manage volumes

Unlike a bind mount, you can create and manage volumes outside the scope of any container.

Let's create a volume

```
$ docker volume create my-vol
```

List volumes:

```
$ docker volume ls

local                my-vol
```

Inspect a volume by looking at the Mount section in the *docker volume inspect*

```
$ docker volume inspect my-vol
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

Remove a volume

```
$ docker volume rm my-vol
```

3.1.3 Populate a volume using a container

This example starts an `nginx` container and populates the new volume `nginx-vol` with the contents of the container's `/var/log/nginx` directory, which is where Nginx stores its log files.

```
$ docker run -d -p 8891:80 --name=nginxtest --mount source=nginx-vol,target=/var/log/
↪nginx nginx:latest
```

So, we now have a copy of Nginx running inside a Docker container on our machine, and our host machine's port 5000 maps directly to that copy of Nginx's port 80. Let's use curl to do a quick test request:

```
$ curl localhost:8891
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

You'll get a screenful of HTML back from Nginx showing that Nginx is up and running. But more interestingly, if you look in the `nginx-vol` volume on the host machine and take a look at the `access.log` file you'll see a log message from Nginx showing our request.

```
cat nginx-vol/_data/access.log
```

Use `docker inspect nginx-vol` to verify that the volume was created and mounted correctly. Look for the Mounts section:

```
"Mounts": [
            {
                "Type": "volume",
                "Name": "nginx-vol",
                "Source": "/var/lib/docker/volumes/nginx-vol/_data",
                "Destination": "/var/log/nginx",
                "Driver": "local",
                "Mode": "z",
                "RW": true,
                "Propagation": ""
            }
        ],
```

This shows that the mount is a volume, it shows the correct source and destination, and that the mount is read-write.

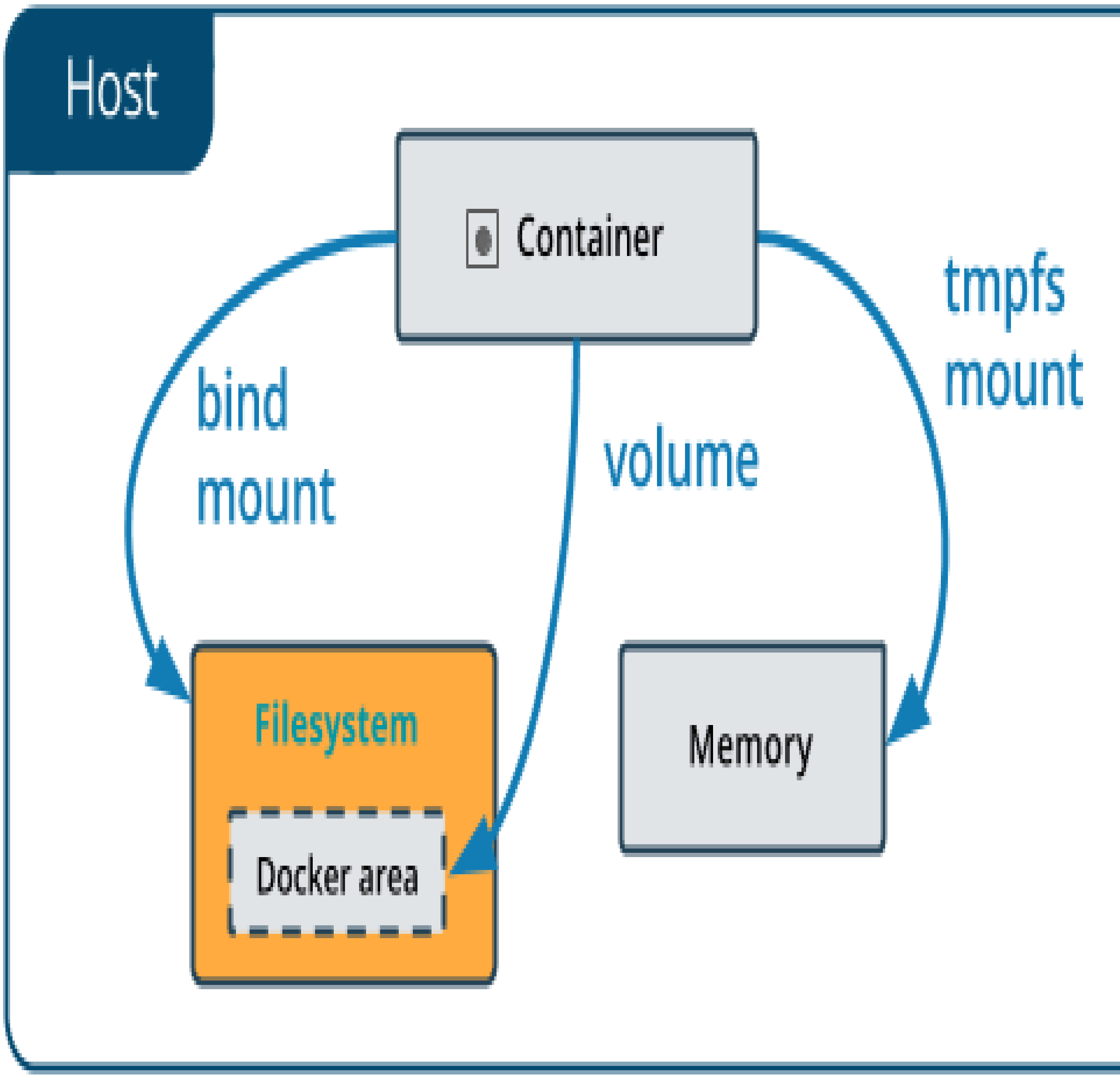
After running either of these examples, run the following commands to clean up the containers and volumes.

```
$ docker stop nginxtest

$ docker rm nginxtest

$ docker volume rm nginx-vol
```

11.3.2 3.2 Bind mounts

Bind mounts: When you use a bind mount, a file or directory on the host machine is mounted into a container.

Tip: If you are developing new Docker applications, consider using named **volumes** instead. You can't use Docker CLI commands to directly manage bind mounts.



Warning: One side effect of using bind mounts, for better or for worse, is that you can change the host filesystem via processes running in a container, including creating, modifying, or deleting important system files or directories. This is a powerful ability which can have security implications, including impacting non-Docker processes on the host system.

If you use `--mount` to bind-mount a file or directory that does not yet exist on the Docker host, Docker does not automatically create it for you, but generates an error.

3.2.1 Start a container with a bind mount

```
$ mkdir data

$ docker run -d -p 8891:80 --name devtest --mount type=bind,source="$(pwd)"/data,
↪target=/var/log/nginx nginx:latest
```

Use *docker inspect devtest* to verify that the bind mount was created correctly. Look for the "Mounts" section

This shows that the mount is a bind mount, it shows the correct source and target, it shows that the mount is read-write, and that the propagation is set to rprivate.

Stop the container:

```
$ docker rm -f devtest
```

3.2.2 Use a read-only bind mount

For some development applications, the container needs to write into the bind mount, so changes are propagated back to the Docker host. At other times, the container only needs read access.

This example modifies the one above but mounts the directory as a read-only bind mount, by adding `ro` to the (empty by default) list of options, after the mount point within the container. Where multiple options are present, separate them by commas.

```
$ docker run -d -p 8891:80 --name devtest --mount type=bind,source="$(pwd)"/data,
↪target=/var/log/nginx,readonly nginx:latest
```

Use `docker inspect devtest` to verify that the bind mount was created correctly. Look for the Mounts section:

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/Users/upendra_35/Documents/git.repos/flask-app/data",
        "Destination": "/var/log/nginx",
        "Mode": "",
        "RW": false,
        "Propagation": "rprivate"
    }
],
```

Stop the container:

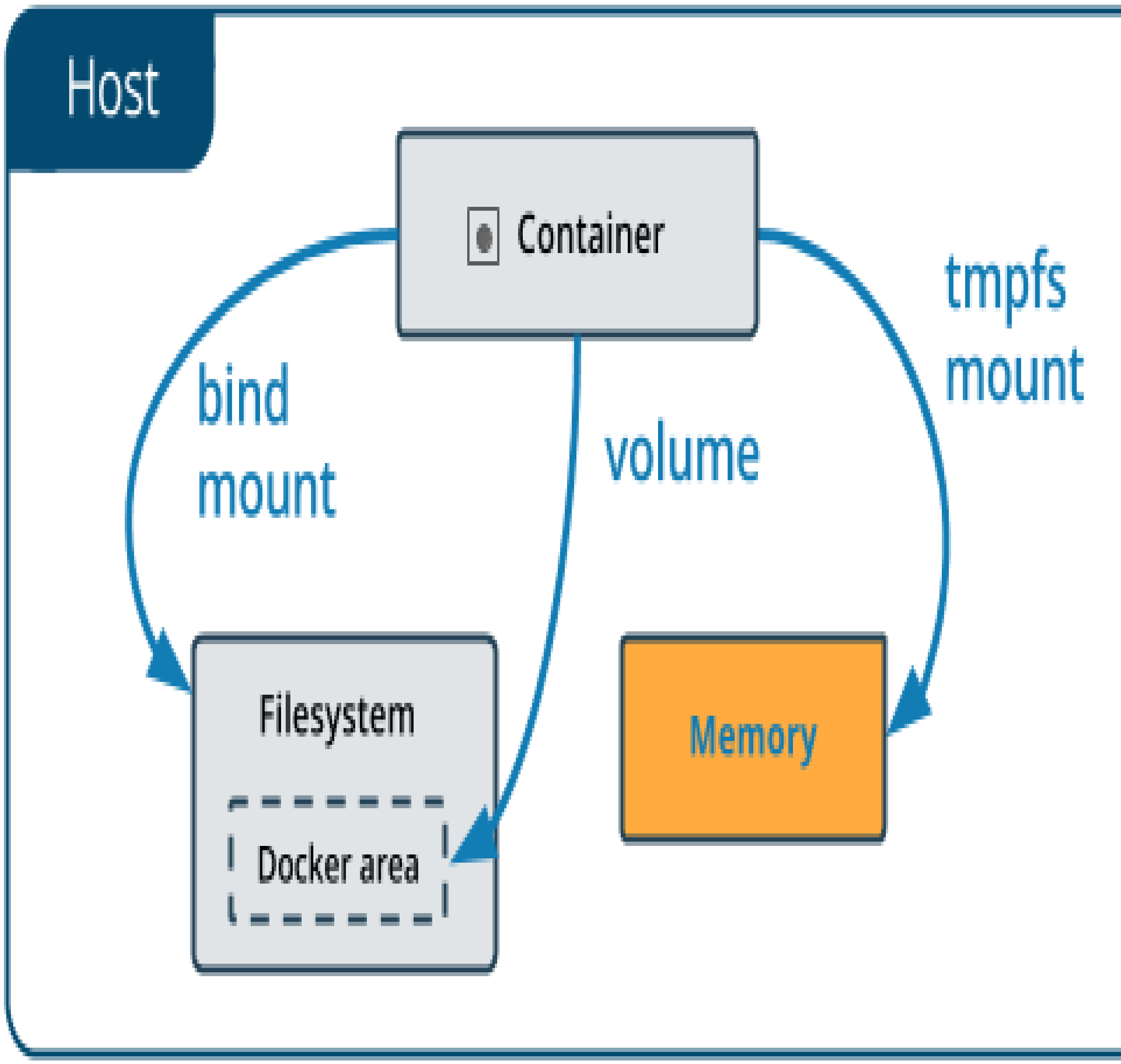
```
$ docker rm -f devtest
```

Remove the volume:

```
$ docker volume rm devtest
```

11.3.3 3.3 tmpfs

tmpfs mounts: A tmpfs mount is not persisted on disk, either on the Docker host or within a container. It can be used by a container during the lifetime of the container, to store non-persistent state or sensitive information. For instance, internally, swarm services use tmpfs mounts to mount secrets into a service's containers.



Volumes and **bind mounts** are mounted into the container's filesystem by default, and their contents are stored on the host machine. There may be cases where you do not want to store a container's data on the host machine, but you also don't want to write the data into the container's writable layer, for performance or security reasons, or if the data relates to non-persistent application state. An example might be a temporary one-time password that the container's application creates and uses as-needed. To give the container access to the data without writing it anywhere permanently, you can use a tmpfs mount, which is only stored in the host machine's memory (or swap, if memory is low). When the container stops, the tmpfs mount is removed. If a container is committed, the tmpfs mount is not saved.

```
$ docker run -d -p 8891:80 --name devtest --mount type=tmpfs,target=/var/log/nginx␣
↪nginx:latest
```

Use *docker inspect devtest* to verify that the bind mount was created correctly. Look for the Mounts section:

```
$ docker inspect devtest

"Mounts": [
            {
                "Type": "tmpfs",
                "Source": "",
                "Destination": "/var/log/nginx",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ],
```

You can see from the above output that the `Source` filed is empty which indicates that the contents are not avaible on Docker host or host file system.

Stop the container:

```
$ docker rm -f devtest
```

Remove the volume:

```
$ docker volume rm devtest
```

11.4 4. Docker Compose for multi container apps

Docker Compose is a tool for defining and running your multi-container Docker applications.

Main advantages of Docker compose include:

- Your applications can be defined in a YAML file where all the options that you used in `docker run` are now defined (Reproducibility).
- It allows you to manage your application as a single entity rather than dealing with individual containers (Simplicity).

Let's now create a simple web app with Docker Compose using Flask (which you already seen before) and Redis. We will end up with a Flask container and a Redis container all on one host.

Note: The code for the above compose example is available here

1. You'll need a directory for your project on your host machine:

```
$ mkdir compose_flask && cd compose_flask
```

2. Add the following to *requirements.txt* inside *compose_flask* directory:

```
flask
redis
```

3. Copy and paste the following code into a new file called *app.py* inside *compose_flask* directory:

```
from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host='redis', port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'This Compose/Flask demo has been viewed %s time(s).' % redis.get('hits')

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

4. Create a Dockerfile with the following code inside `compose_flask` directory:

```
FROM python:2.7
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD python app.py
```

5. Add the following code to a new file, `docker-compose.yml`, in your project directory:

```
version: '2'
services:
    web:
        restart: always
        build: .
        ports:
            - "8888:5000"
        volumes:
            - .:/code
        depends_on:
            - redis
    redis:
        restart: always
        image: redis
```

A brief explanation of `docker-compose.yml` is as below:

- `restart: always` means that it will restart whenever it fails.
- We define two services, **web** and **redis**.
- The web service builds from the Dockerfile in the current directory.
- Forwards the container's exposed port (5000) to port 8888 on the host.
- Mounts the project directory on the host to /code inside the container (allowing you to modify the code without having to rebuild the image).
- `depends_on` links the web service to the Redis service.
- The redis service uses the latest Redis image from Docker Hub.

Note: Docker for Mac and Docker Toolbox already include Compose along with other Docker apps, so Mac users do not need to install Compose separately. Docker for Windows and Docker Toolbox already include Compose along with other Docker apps, so most Windows users do not need to install Compose separately.

For Linux users

```
sudo curl -L https://github.com/docker/compose/releases/download/1.19.0/docker-
↪compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

---

5. Build and Run with `docker-compose up -d` command

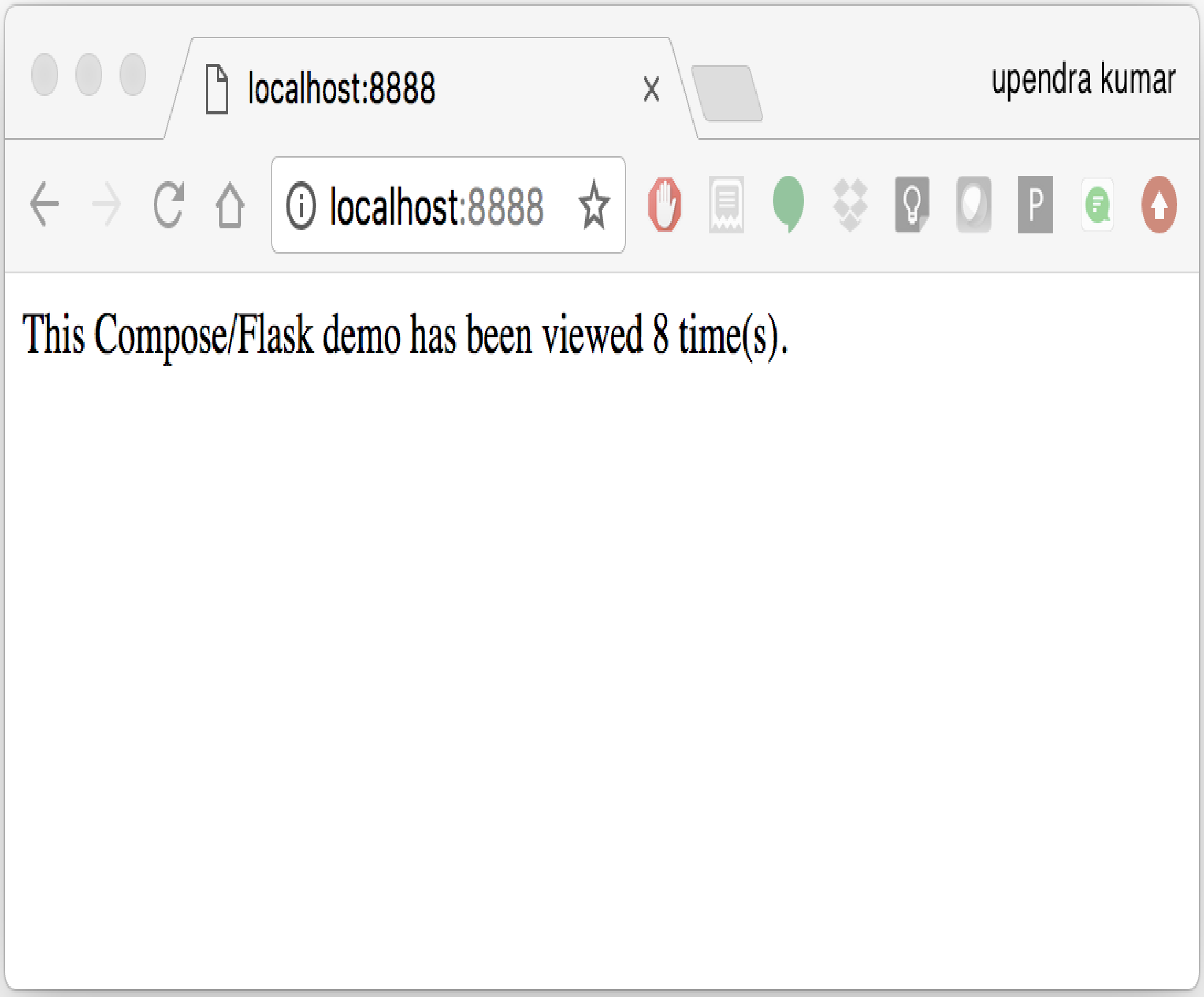
```
$ docker-compose up -d

Building web
Step 1/5 : FROM python:2.7
2.7: Pulling from library/python
f49cf87b52c1: Already exists
7b491c575b06: Already exists
b313b08bab3b: Already exists
51d6678c3f0e: Already exists
09f35bd58db2: Already exists
f7e0c30e74c6: Pull complete
c308c099d654: Pull complete
339478b61728: Pull complete
Digest: sha256:8cb593cb9cd1834429f0b4953a25617a8457e2c79b3e111c0f70bffd21acc467
Status: Downloaded newer image for python:2.7
 ---> 9e92c8430ba0
Step 2/5 : ADD . /code
 ---> 746bcecfc3c9
Step 3/5 : WORKDIR /code
 ---> c4cf3d6cb147
Removing intermediate container 84d850371a36
Step 4/5 : RUN pip install -r requirements.txt
 ---> Running in d74c2e1cfbf7
Collecting flask (from -r requirements.txt (line 1))
  Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)
Collecting redis (from -r requirements.txt (line 2))
  Downloading redis-2.10.6-py2.py3-none-any.whl (64kB)
Collecting itsdangerous>=0.21 (from flask->-r requirements.txt (line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting Jinja2>=2.4 (from flask->-r requirements.txt (line 1))
  Downloading Jinja2-2.10-py2.py3-none-any.whl (126kB)
Collecting Werkzeug>=0.7 (from flask->-r requirements.txt (line 1))
  Downloading Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
Collecting click>=2.0 (from flask->-r requirements.txt (line 1))
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->flask->-r requirements.txt (line 1))
  Downloading MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/fc/a8/66/
↪24d655233c757e178d45dea2de22a04c6d92766abfb741129a
  Running setup.py bdist_wheel for MarkupSafe: started
  Running setup.py bdist_wheel for MarkupSafe: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/88/a7/30/
↪e39a54a87bcbe25308fa3ca64e8ddc75d9b3e5afa21ee32d57
```

(continues on next page)

(continued from previous page)

```
Successfully built itsdangerous MarkupSafe
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug, click,␣
↪flask, redis
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 click-6.7 flask-0.
↪12.2 itsdangerous-0.24 redis-2.10.6
 ---> 5cc574ff32ed
Removing intermediate container d74c2e1cfbf7
Step 5/5 : CMD python app.py
 ---> Running in 3ddb7040e8be
 ---> e911b8e8979f
Removing intermediate container 3ddb7040e8be
Successfully built e911b8e8979f
Successfully tagged composeflask_web:latest
```

And that's it! You should be able to see the Flask application running on `http://localhost:8888` or `<ipaddress>:8888`



This Compose/Flask demo has been viewed 8 time(s).

11.4.1 Exercise 2 (10 mins)

- Change the greeting in `app.py` and save it. For example, change the `This Compose/Flask demo has been viewed` message to `This Container Camp Workshop demo has been viewed`
- Refresh the *app* in your browser. What do you see now?
- Create a automatic build for `compose-flask` project directory

- Share your Dockerhub link url on Slack

11.5 5. Improving your data science workflow using Docker containers (Containerized Data Science)

For a data scientist, running a container that is already equipped with the libraries and tools needed for a particular analysis eliminates the need to spend hours debugging packages across different environments or configuring custom environments.

But why Set Up a Data Science Environment in a Container?

- One reason is speed. We want data scientists using our platform to launch a Jupyter or RStudio session in minutes, not hours. We also want them to have that fast user experience while still working in a governed, central architecture (rather than on their local machines).
- Containerization benefits both data science and IT/technical operations teams. In the DataScience.com Platform, for instance, we allow IT to configure environments with different languages, libraries, and settings in an admin dashboard and make those images available in the dropdown menu when a data scientist launches a session. These environments can be selected for any run, session, scheduled job, or API. (Or you don't have to configure anything at all. We provide plenty of standard environment templates to choose from.)
- Ultimately, containers solve a lot of common problems associated with doing data science work at the enterprise level. They take the pressure off of IT to produce custom environments for every analysis, standardize how data scientists work, and ensure that old code doesn't stop running because of environment changes. To start using containers and our library of curated images to do collaborative data science work, request a demo of our platform today.
- Configuring a data science environment can be a pain. Dealing with inconsistent package versions, having to dive through obscure error messages, and having to wait hours for packages to compile can be frustrating. This makes it hard to get started with data science in the first place, and is a completely arbitrary barrier to entry.

Thanks to the rich ecosystem, there are already several readily available images for the common components in data science pipelines. Here are some Docker images to help you quickly spin up your own data science pipeline:

- MySQL
- Postgres
- Redmine
- MongoDB
- Hadoop
- Spark
- Zookeeper
- Kafka
- Cassandra
- Storm
- Flink
- R

Motivation: Say you want to play around with some cool data science libraries in Python or R but what you don't want to do is spend hours on installing Python or R, working out what libraries you need, installing each and every one and then messing around with the tedium of getting things to work just right on your version of

Linux/Windows/OSX/OS9—well this is where Docker comes to the rescue! With Docker we can get a Jupyter 'Data Science' notebook stack up and running in no time at all. Let's get started! We will see few examples of thse in the following sections…

Note: The above code can be found in this github

1. Launch a Jupyter notebook conatiner

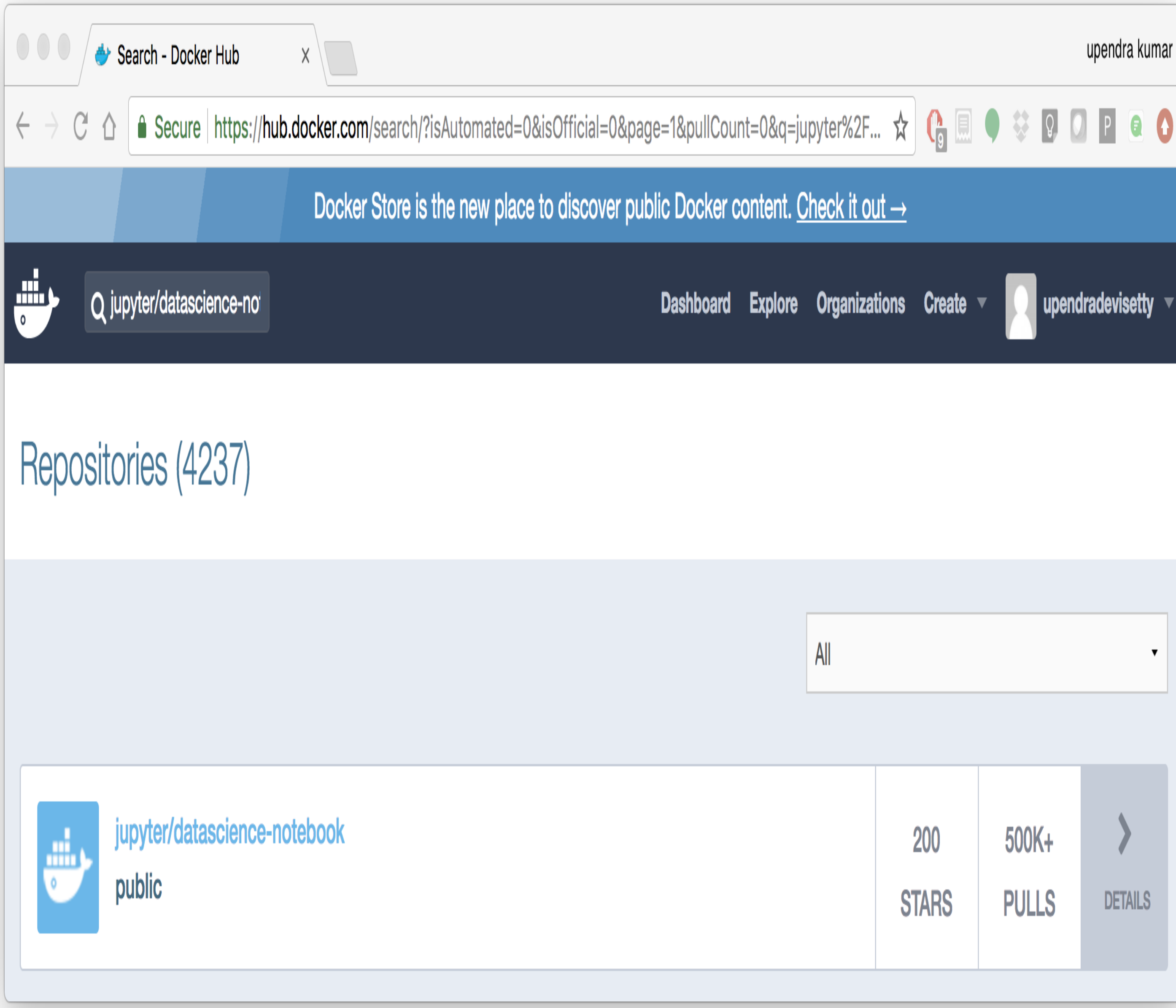
Docker allows us to run a 'ready to go' Jupyter data science stack in what's known as a container:

1.1 Create a *docker-compose.yml* file

```
$ mkdir jn && cd jn
```

```
version: '2'

services:
  datascience-notebook:
    image: jupyter/datascience-notebook
    volumes:
      - .:/data
    ports:
      - 8888:8888
    container_name:   datascience-notebook-container
```

Note: The `jupyter/datascience-notebook` image can be found on dockerhub


Search - Docker Hub
upendra kumar
Secure https://hub.docker.com/search/?isAutomated=0&isOfficial=0&page=1&pullCount=0&q=jupyter%2F...
Docker Store is the new place to discover public Docker content. Check it out →
jupyter/datascience-no
Dashboard
Explore
Organizations
Create
upendradevisetty
Repositories (4237)
All
jupyter/datascience-notebook
public
200
STARS
500K+
PULLS
DETAILS

1.2 Run container using docker-compose file

```
$ docker-compose up
Creating datascience-notebook-container ...
Creating datascience-notebook-container ... done
Attaching to datascience-notebook-container
datascience-notebook-container | Execute the command: jupyter notebook
datascience-notebook-container | [I 08:44:31.312 NotebookApp] Writing notebook server␣
↪cookie secret to /home/jovyan/.local/share/jupyter/runtime/notebook_cookie_secret
```

(continues on next page)

(continued from previous page)

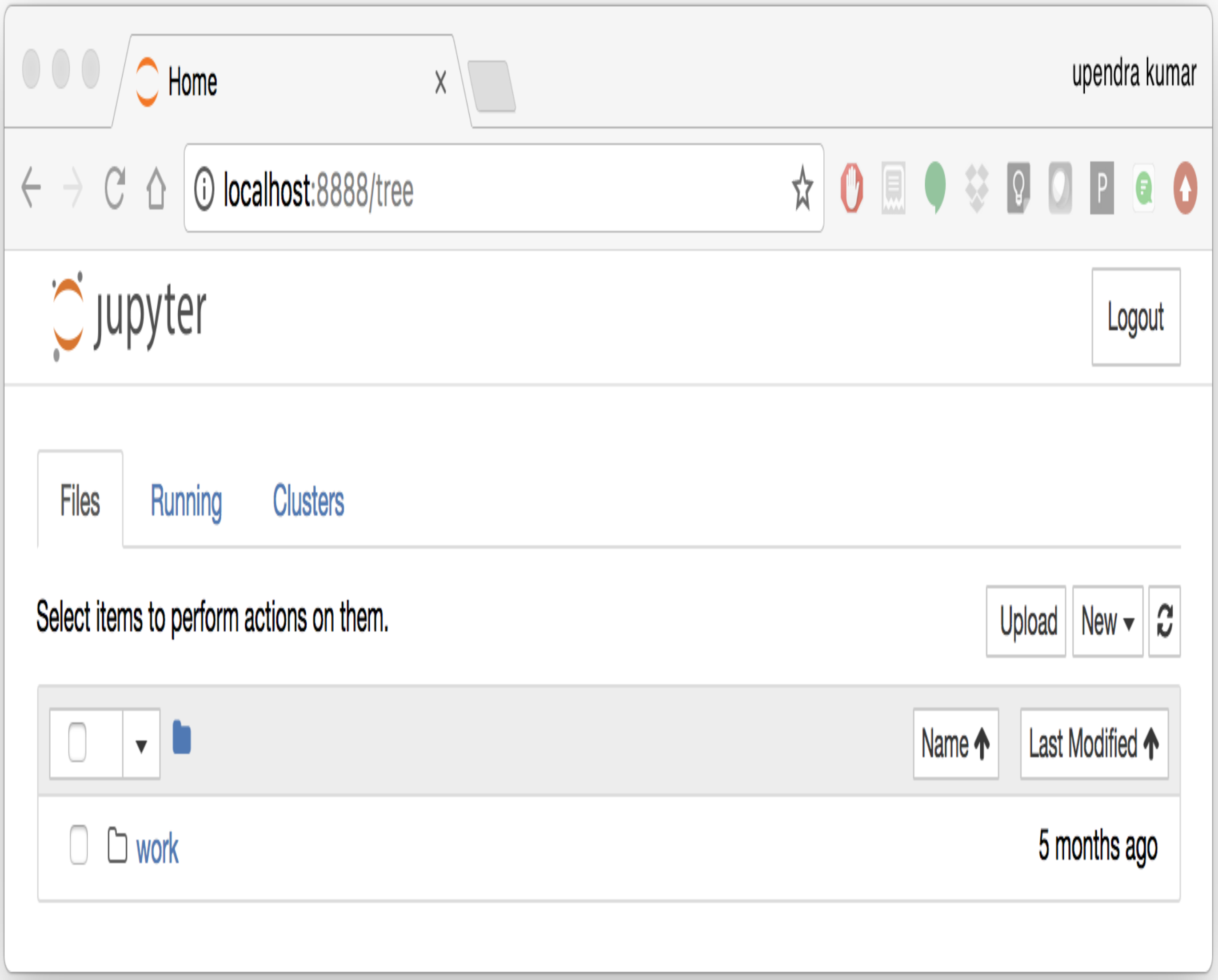
```
datascience-notebook-container | [W 08:44:31.332 NotebookApp] WARNING: The notebook␣
↪server is listening on all IP addresses and not using encryption. This is not     ␣
↪recommended.
datascience-notebook-container | [I 08:44:31.370 NotebookApp] JupyterLab alpha␣
↪preview extension loaded from /opt/conda/lib/python3.6/site-packages/jupyterlab
datascience-notebook-container | JupyterLab v0.27.0
datascience-notebook-container | Known labextensions:
datascience-notebook-container | [I 08:44:31.373 NotebookApp] Running the core␣
↪application with no additional extensions or settings
datascience-notebook-container | [I 08:44:31.379 NotebookApp] Serving notebooks from␣
↪local directory: /home/jovyan
datascience-notebook-container | [I 08:44:31.379 NotebookApp] 0 active kernels
datascience-notebook-container | [I 08:44:31.379 NotebookApp] The Jupyter Notebook is␣
↪running at: http://[all ip addresses on your      system]:8888/?
↪token=dfb50de6c1da091fd62336ac52cdb88de5fe339eb0faf478
datascience-notebook-container | [I 08:44:31.379 NotebookApp] Use Control-C to stop␣
↪this server and shut down all kernels (twice to skip confirmation).
datascience-notebook-container | [C 08:44:31.380 NotebookApp]
datascience-notebook-container |
datascience-notebook-container |     Copy/paste this URL into your browser when you␣
↪connect for the first time,
datascience-notebook-container |     to login with a token:
datascience-notebook-container |         http://localhost:8888/?
↪token=dfb50de6c1da091fd62336ac52cdb88de5fe339eb0faf478
```

The last line is a URL that we need to copy and paste into our browser to access our new Jupyter stack:

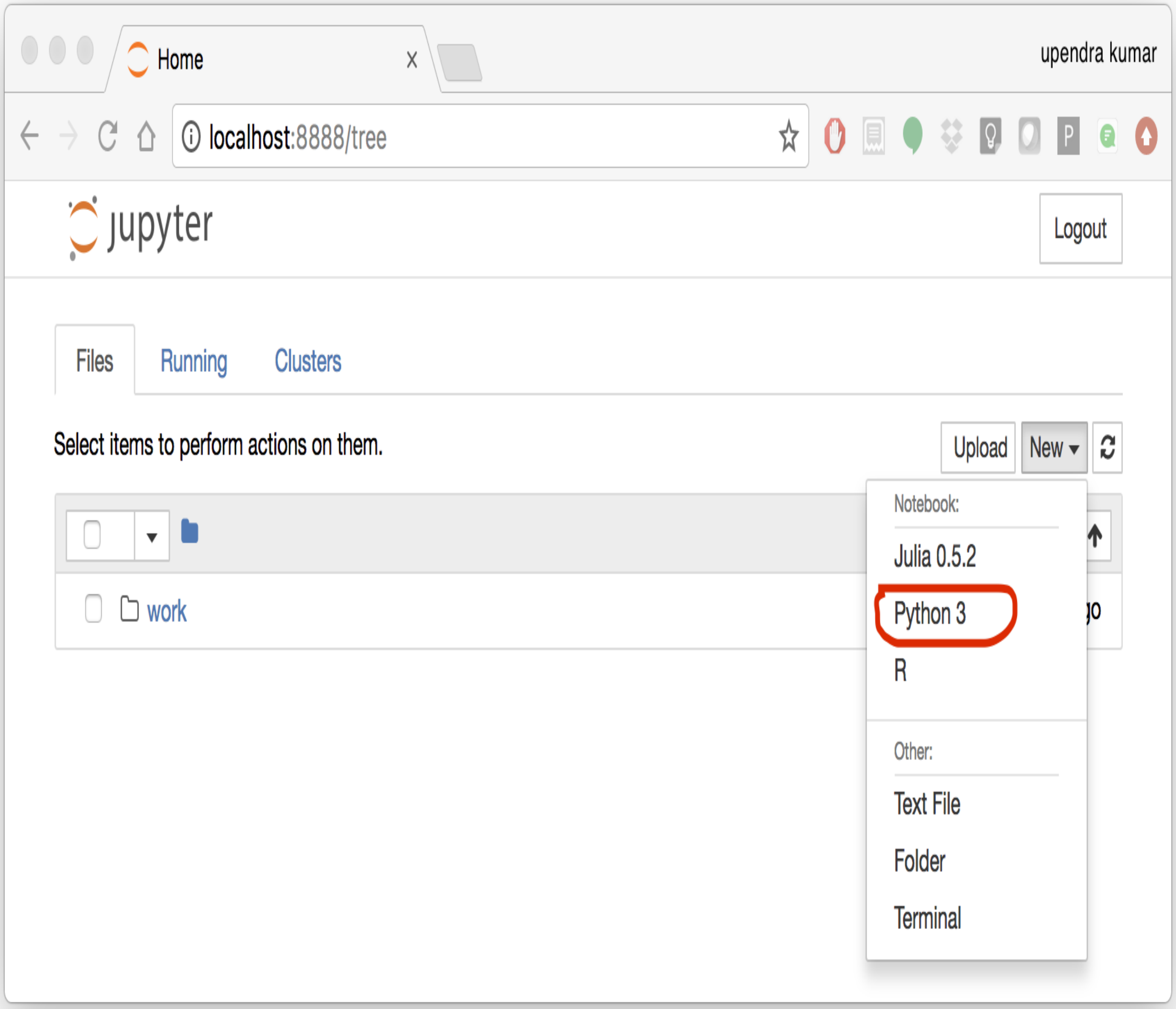
```
http://localhost:8888/?token=dfb50de6c1da091fd62336ac52cdb88de5fe339eb0faf478
```

Warning: Do not copy and paste the above URL in your browser as this URL is specific to my environment.

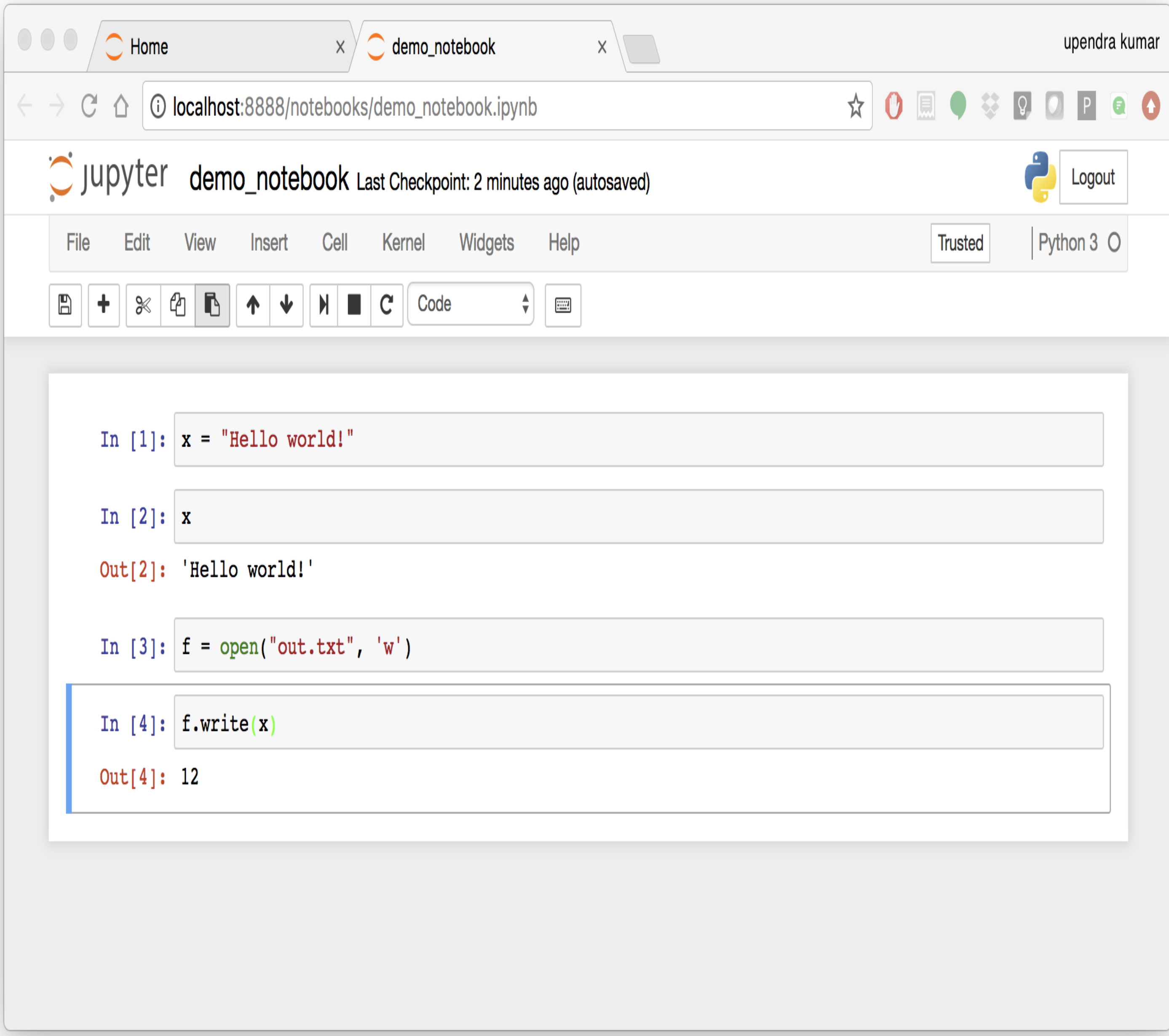
Once you've done that you should be greeted by your very own containerised Jupyter service!



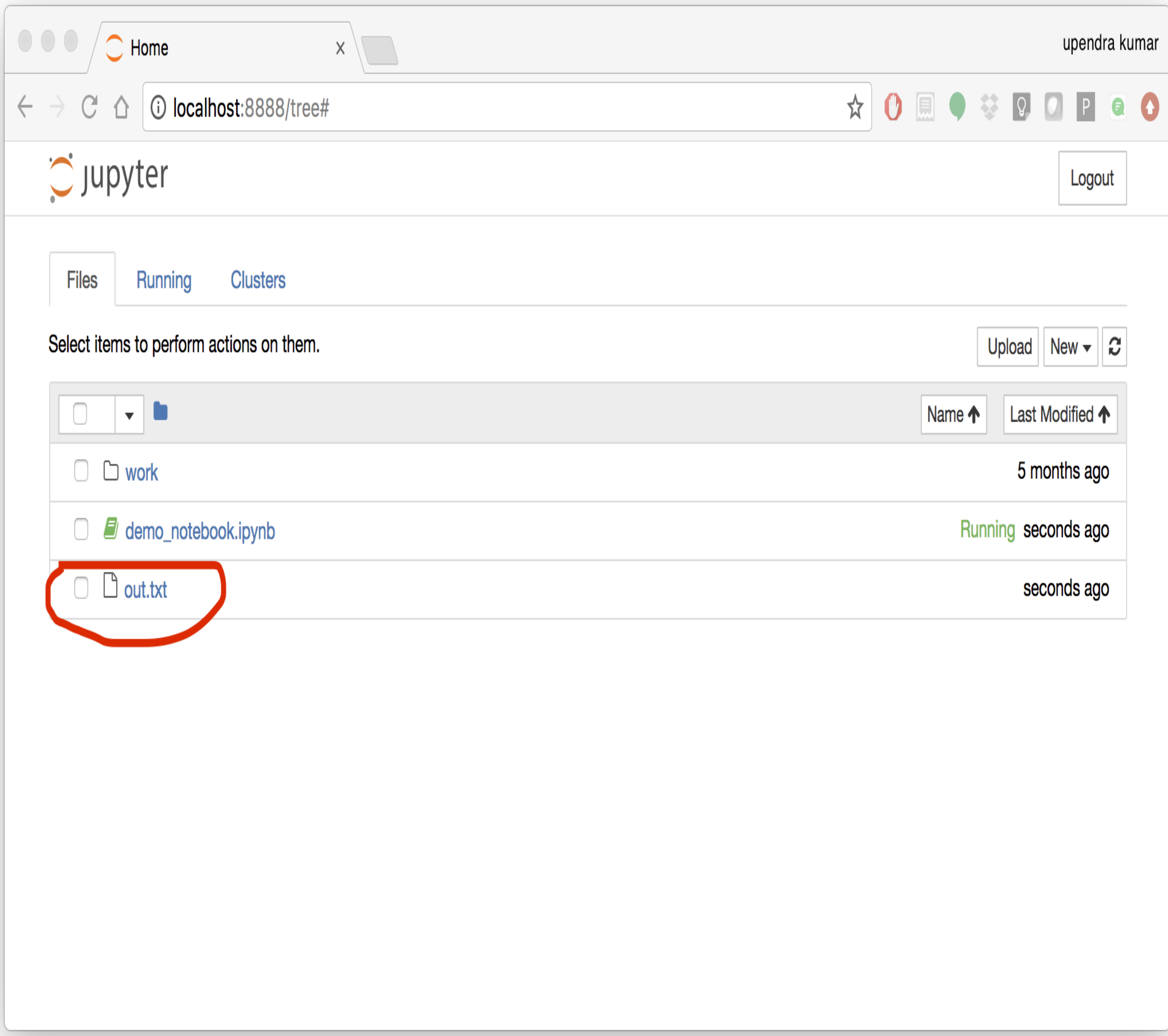
To create your first notebook, drill into the work directory and then click on the ‘New’ button on the right hand side and choose ‘Python 3’ to create a new Python 3 based Notebook.



Now you can write your python code. Here is an example



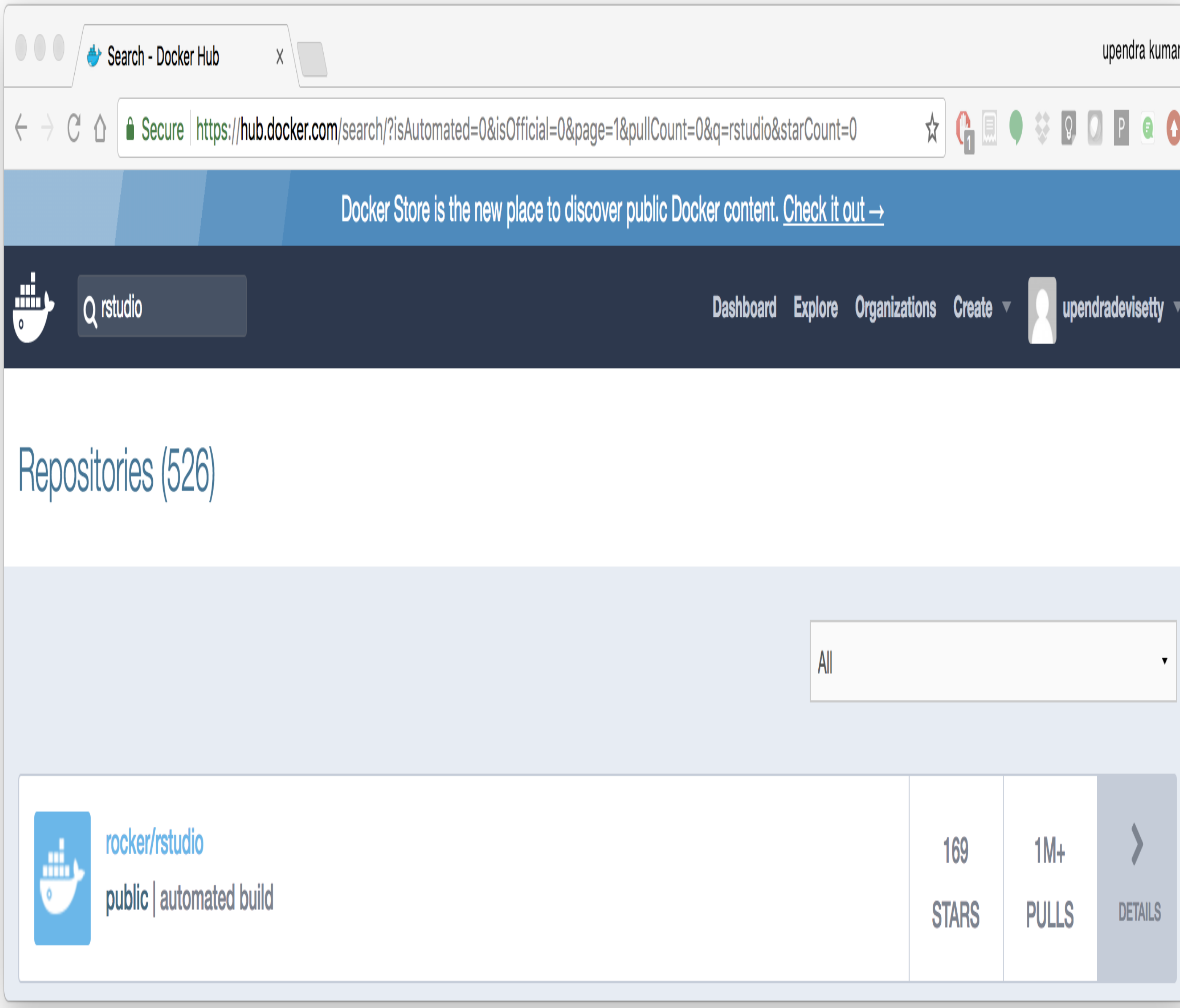
```
In [1]: x = "Hello world!"

In [2]: x

Out[2]: 'Hello world!'

In [3]: f = open("out.txt", 'w')

In [4]: f.write(x)

Out[4]: 12
```



To shut down the container once you're done working, simply hit Ctrl-C in the terminal/command prompt. Your work will all be saved on your actual machine in the path we set in our Docker compose file. And there you have it—a quick and easy way to start using Jupyter notebooks with the magic of Docker.

2. Launch a R-Studio container

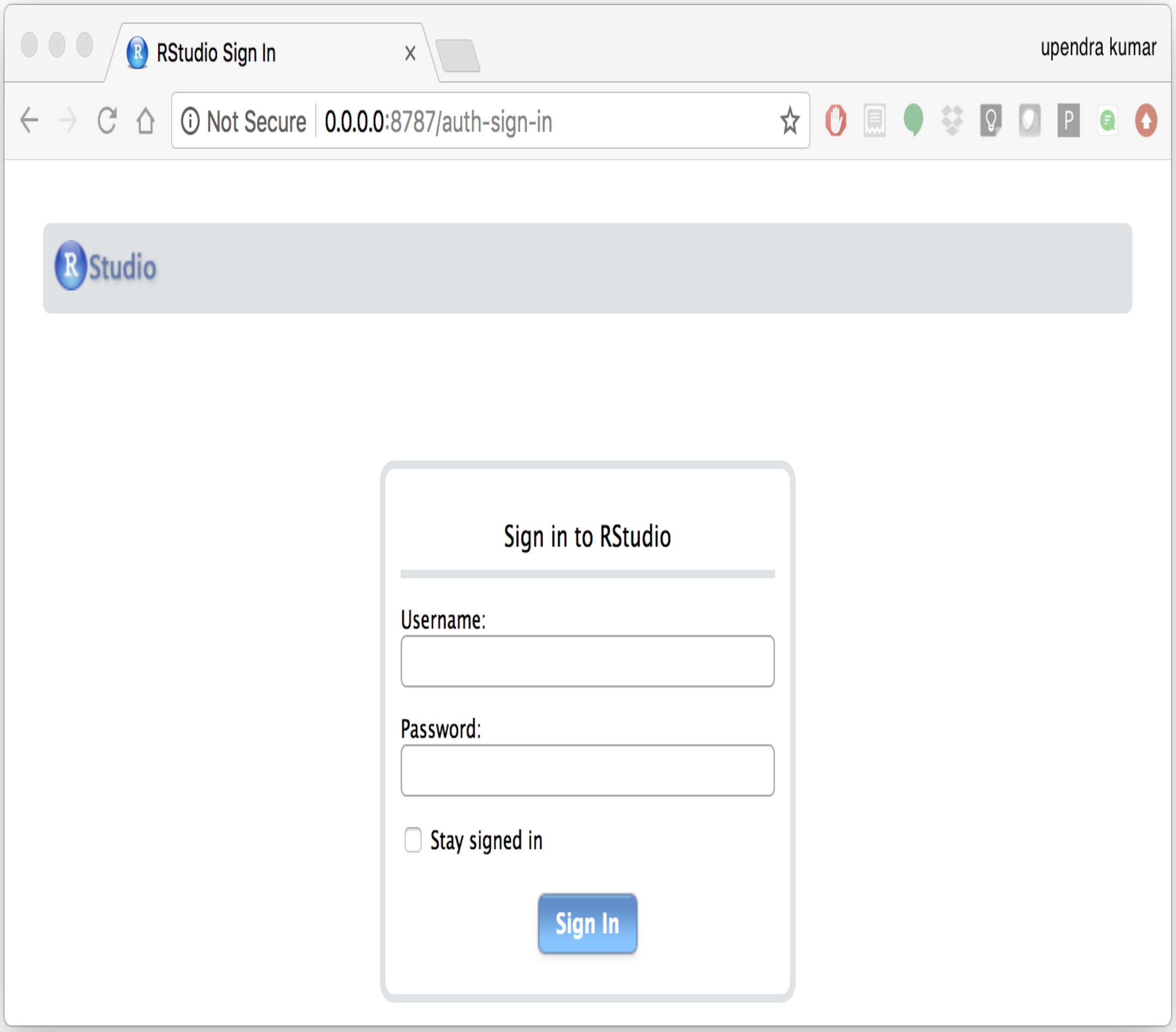
Next, we will see a Docker image from Rocker which will allow us to run RStudio inside the container and has many useful R packages already installed.


Search - Docker Hub
upendra kumar
Secure https://hub.docker.com/search/?isAutomated=0&isOfficial=0&page=1&pullCount=0&q=rstudio&starCount=0
Docker Store is the new place to discover public Docker content. Check it out →
rstudio
Dashboard
Explore
Organizations
Create
upendradevisetty
Repositories (526)
All
rocker/rstudio
public | automated build
169
STARS
1M+
PULLS
DETAILS

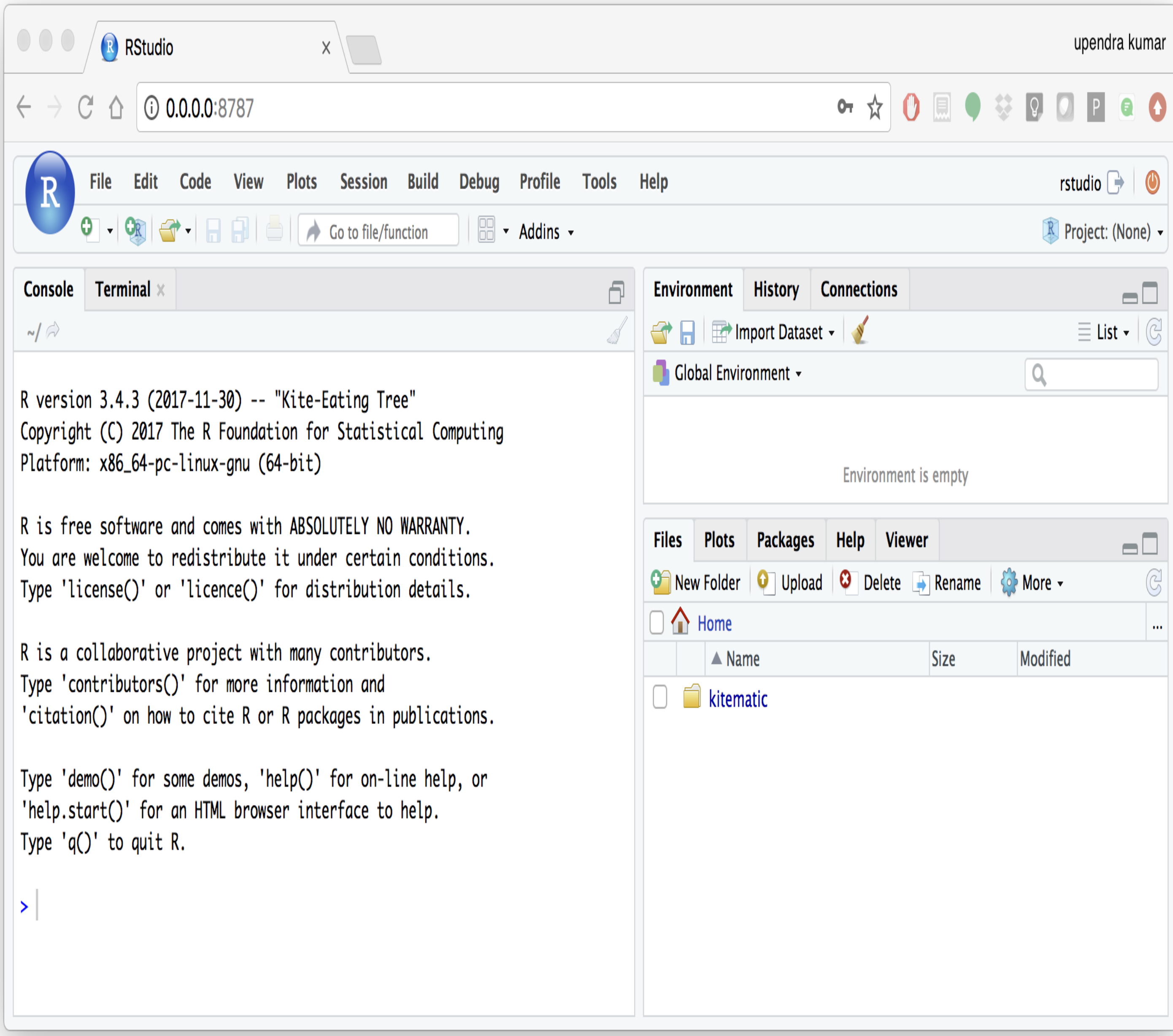
```
$ docker run --rm -d -p 8787:8787 rocker/rstudio:3.4.3
```

Note: `–rm` ensures that when we quit the container, the container is deleted. If we did not do this, everytime we run a container, a version of it will be saved to our local computer. This can lead to the eventual wastage of a lot of disk space until we manually remove these containers.

The command above will lead RStudio-Server to launch invisibly. To connect to it, open a browser and enter http://localhost:8787, or <ipaddress>:8787 on cloud



Enter `rstudio` as username and password. Finally Rstudio shows up and you can run your R command from here



```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

3. Machine learning using Docker

In this simple example we'll take a sample dataset of fruits metrics (like size, weight, texture) labelled apples and oranges. Then we can predict the fruit given a new set of fruit metrics using scikit-learn's decision tree

You can find the above code in this github repo

1. Create a directory that consists of all the files

```
$ mkdir scikit_docker && cd scikit_docker
```

2. Create `requirements.txt` file—Contains python modules and has nothing to do with Docker inside the folder - `scikit_docker`.

```
numpy
scipy
scikit-learn
```

3. Create a file called `app.py` inside the folder— `scikit_docker`

```
from sklearn import tree
#DataSet
#[size,weight,texture]
X = [[181, 80, 44], [177, 70, 43], [160, 60, 38], [154, 54, 37],[166, 65, 40], [190,␣
↪90, 47], [175, 64, 39], [177, 70, 40], [159, 55, 37], [171, 75, 42], [181, 85, 43]]

Y = ['apple', 'apple', 'orange', 'orange', 'apple', 'apple', 'orange', 'orange',
↪'orange', 'apple', 'apple']

#classifier - DecisionTreeClassifier
clf_tree = tree.DecisionTreeClassifier();
clf_tree = clf_tree.fit(X,Y);

#test_data
test_data = [[190,70,42],[172,64,39],[182,80,42]];

#prediction
prediction_tree = clf_tree.predict(test_data);

# Write output to a file
with open("output.txt", 'w') as fh_out:
        fh_out.write("Prediction of DecisionTreeClassifier:")
        fh_out.write(str(prediction_tree))
```

4. Create a Dockerfile that contains all the instructions for building a Docker image inside the project directory

```
# Use an official Python runtime as a parent image
FROM python:3.6-slim
MAINTAINER Upendra Devisetty <upendra@cyverse.org>
LABEL Description "This Dockerfile is used to build a scikit-learn's decision tree␣
↪image"

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

5. Create a Docker compose YAML file

```
version: '2'
services:
    datasci:
        build: .
        volumes:
            - .:/app
```

5. Now Build and Run the Docker image using *docker-compose up* command to predict the fruit given a new set of fruit metrics

```
$ docker-compose up
Building datasci
Step 1/8 : FROM python:3.6-slim
 ---> dc41c0491c65
Step 2/8 : MAINTAINER Upendra Devisetty <upendra@cyverse.org>
 ---> Running in 95a4da823100
 ---> 7c4d5b78bb0a
Removing intermediate container 95a4da823100
Step 3/8 : LABEL Description "This Dockerfile is used to build a scikit-learn's␣
↪decision tree image"
 ---> Running in e8000ae57a7d
 ---> d872e29971e3
Removing intermediate container e8000ae57a7d
Step 4/8 : WORKDIR /app
 ---> 083eb3e4fb16
Removing intermediate container c965871286f9
Step 5/8 : ADD . /app
 ---> 82b1dbdbe759
Step 6/8 : RUN pip install -r requirements.txt
 ---> Running in 3c82f7d5dd95
Collecting numpy (from -r requirements.txt (line 1))
  Downloading numpy-1.14.0-cp36-cp36m-manylinux1_x86_64.whl (17.2MB)
Collecting scipy (from -r requirements.txt (line 2))
  Downloading scipy-1.0.0-cp36-cp36m-manylinux1_x86_64.whl (50.0MB)
Collecting scikit-learn (from -r requirements.txt (line 3))
  Downloading scikit_learn-0.19.1-cp36-cp36m-manylinux1_x86_64.whl (12.4MB)
Installing collected packages: numpy, scipy, scikit-learn
Successfully installed numpy-1.14.0 scikit-learn-0.19.1 scipy-1.0.0
 ---> 3d402c23203f
Removing intermediate container 3c82f7d5dd95
Step 7/8 : ENV NAME World
 ---> Running in d0468b521e81
 ---> 9cd31e8e7c95
Removing intermediate container d0468b521e81
Step 8/8 : CMD python app.py
 ---> Running in 051bd2235697
 ---> 36bb4c3d9183
Removing intermediate container 051bd2235697
Successfully built 36bb4c3d9183
Successfully tagged scikitdocker_datasci:latest
WARNING: Image for service datasci was built because it did not already exist. To␣
↪rebuild this image you must use `docker-compose build` or `docker-compose up --
↪build`.
Creating scikitdocker_datasci_1 ...
Creating scikitdocker_datasci_1 ... done
Attaching to scikitdocker_datasci_1
scikitdocker_datasci_1 exited with code 0
```

Use `docker-compose rm` to remove the container after docker-compose finish running

```
docker-compose rm
Going to remove scikitdocker_datasci_1
Are you sure? [yN] y
Removing scikitdocker_datasci_1 ... done
```

You will find the ouput file in the `scikit_docker` folder with the following contents

```
$ cat output.txt
Prediction of DecisionTreeClassifier:['apple' 'orange' 'apple']
```

CHAPTER 12

Introduction to Singularity

12.1 1. Prerequisites

There are no specific skills needed for this tutorial beyond a basic comfort with the command line and using a text editor. Prior experience developing web applications could be helpful but is not required.

Note:

Important: Docker and Singularity are friends but they have distinct differences.

Singularity Related Resources

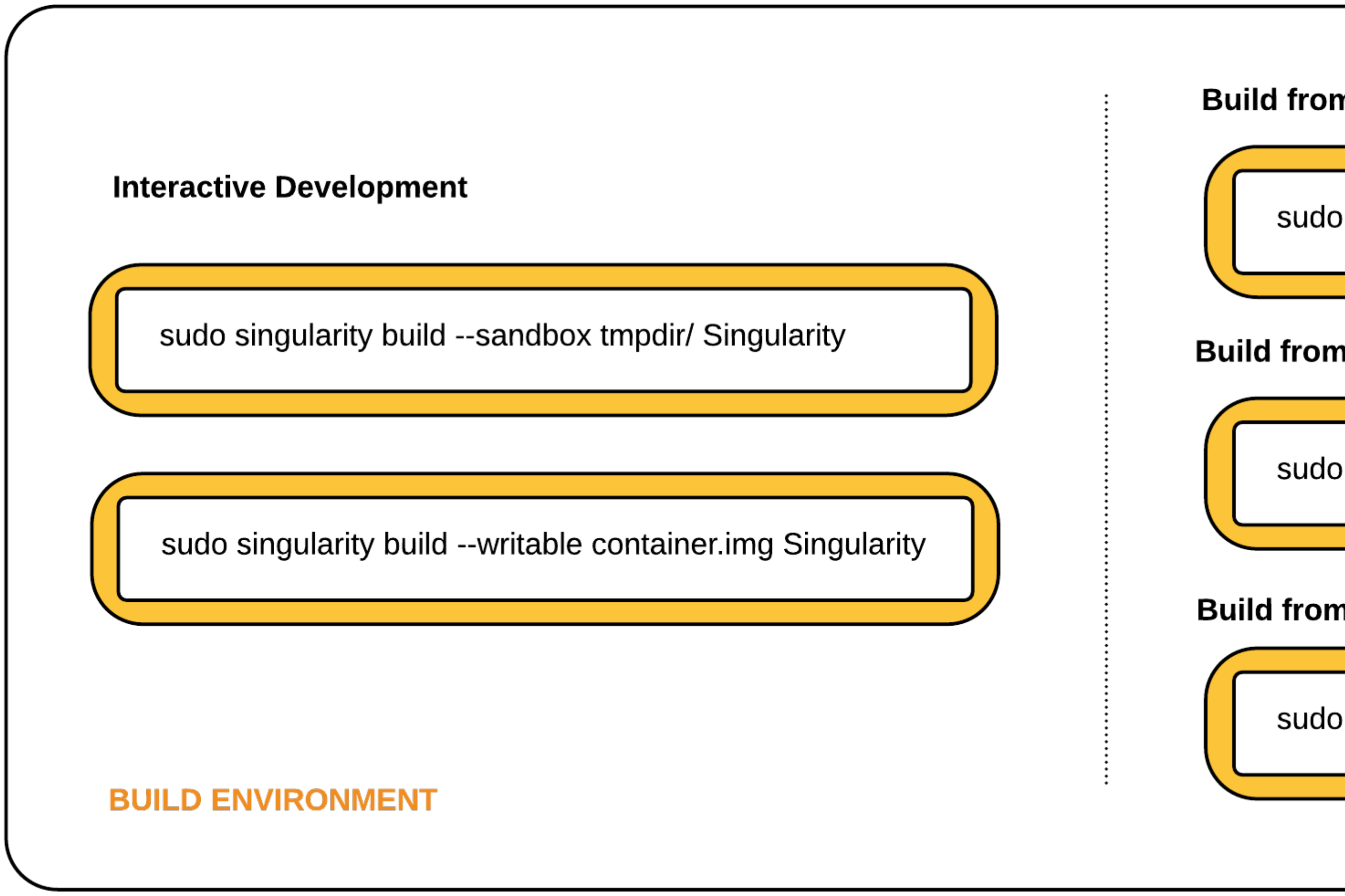
Docker:

- Inside a Docker container the user has escalated privileges, effectively making them root on the host system. This is not supported by most administrators of High Performance Computing (HPC) centers.

Singularity:

- Work on HPC
- Same user inside and outside the container
- User only has root privileges if elevated with *sudo*
- Run (and modify!) existing Docker containers

Singularity uses a 'flow' whereby you can (1) create and modify images on your dev system, (2) build containers using recipes or pulling from repositories, and (3) execute containers on production systems.


Interactive Development
sudo singularity build --sandbox tmpdir/ Singularity
sudo singularity build --writable container.img Singularity
BUILD ENVIRONMENT
Build fron
sudo
Build from
sudo
Build from
sudo

12.2 2. Singularity Installation

Singularity homepage: https://www.sylabs.io/docs/

While Singularity is more likely to be used on a remote system, e.g. HPC or cloud, you may want to develop your own containers first on a local machine or dev system.

12.2.1 Exercise 1 (15-20 mins)

12.2.2 2.1 Setting up your Laptop

To Install Singularity on your laptop or desktop PC follow the instructions from Singularity: (Mac, Windows, Linux)

- running a VM is required on Mac OS X, Singularityware VagrantBox

12.2.3 2.2 HPC

Load the Singularity module on a HPC

If you are interested in working on HPC, you may need to contact your systems administrator and request they install Singularity.

Most HPC systems are running Environment Modules with the simple command *module*. You can check to see what is available:

```
$ module avail
```

If Singularity is installed:

```
$ module load singularity
```

12.2.4 2.3 XSEDE Jetstream / CyVerse Atmosphere Clouds

CyVerse staff have deployed an Ansible playbooks called *ez* installation which includes Singularity that only requires you to type a short line of code.

Start a featured instance on Atmosphere or Jetstream.

Type in the following:

```
$ ezs

* Updating ez singularity and installing singularity (this may take a few minutes,␣
↪coffee break!)
Cloning into '/opt/cyverse-ez-singularity'...
remote: Counting objects: 11, done.
remote: Total 11 (delta 0), reused 0 (delta 0), pack-reused 11
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

12.2.5 2.4 Check Installation

Singularity should now be installed on your laptop or VM, or loaded on the HPC, you can check the installation with:

```
$ singularity pull shub://vsoch/hello-world
Progress |===================================| 100.0%
Done. Container is at: /tmp/vsoch-hello-world-master.simg

$ singularity run vsoch-hello-world-master.simg
RaawwWWWWWRRRR!!
```

View the Singularity help:

```
$ singularity --help

USAGE: singularity [global options...] <command> [command options...] ...

GLOBAL OPTIONS:
    -d|--debug    Print debugging information
    -h|--help     Display usage summary
    -s|--silent   Only print errors
    -q|--quiet    Suppress all normal output
       --version  Show application version
    -v|--verbose  Increase verbosity +1
    -x|--sh-debug Print shell wrapper debugging information

GENERAL COMMANDS:
    help       Show additional help for a command or container
    selftest   Run some self tests for singularity install

CONTAINER USAGE COMMANDS:
    exec       Execute a command within container
    run        Launch a runscript within container
    shell      Run a Bourne shell within container
    test       Launch a testscript within container

CONTAINER MANAGEMENT COMMANDS:
    apps       List available apps within a container
    bootstrap  *Deprecated* use build instead
    build      Build a new Singularity container
    check      Perform container lint checks
    inspect    Display container's metadata
    mount      Mount a Singularity container image
    pull       Pull a Singularity/Docker container to $PWD

COMMAND GROUPS:
    image      Container image command group
    instance   Persistent instance command group


CONTAINER USAGE OPTIONS:
    see singularity help <command>

For any additional help or support visit the Singularity
website: http://singularity.lbl.gov/
```

12.3 3. Downloading Singularity containers

The easiest way to use a Singularity container is to *pull* an existing container from one of the Container Registries maintained by the Singularity group.

12.3.1 Exercise 2 (~10 mins)

12.3.2 3.1: Pulling a Container from Singularity Hub

You can use the *pull* command to download pre-built images from a number of Container Registries, here we'll be focusing on the Singularity-Hub or DockerHub.

Container Registries:

- *shub* - images hosted on Singularity Hub
- *docker* - images hosted on Docker Hub
- *localimage* - images saved on your machine
- *yum* - yum based systems such as CentOS and Scientific Linux
- *debootstrap* - apt based systems such as Debian and Ubuntu
- *arch* - Arch Linux
- *busybox* - BusyBox
- *zypper* - zypper based systems such as Suse and OpenSuse

In this example I am pulling a base Ubuntu container from Singularity-Hub:

```
$ singularity pull shub://singularityhub/ubuntu
```

You can rename the container using the *–name* flag:

```
$ singularity pull --name ubuntu_test.simg shub://singularityhub/ubuntu
```

After your image has finished downloading it should be in the present working directory, unless you specified to download it somewhere else.

```
$ singularity pull --name ubuntu_test.simg shub://singularityhub/ubuntu
Progress |===================================| 100.0%
Done. Container is at: /home/***/ubuntu_test.simg
$ singularity run ubuntu_test.simg
This is what happens when you run the container...
$ singularity shell ubuntu_test.simg
Singularity: Invoking an interactive shell within container...

Singularity ubuntu_test.simg:~> cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04 LTS"
NAME="Ubuntu"
VERSION="14.04, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04 LTS"
```

(continues on next page)

(continued from previous page)

```
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
Singularity ubuntu_test.simg:~>
```

12.3.3 Exercise 2.2: Pulling container from Docker Hub

This example pulls a container from DockerHub

Build to your container by pulling an image from Docker:

```
$ singularity pull docker://ubuntu:16.04
WARNING: pull for Docker Hub is not guaranteed to produce the
WARNING: same image on repeated pull. Use Singularity Registry
WARNING: (shub://) to pull exactly equivalent images.
Docker image path: index.docker.io/library/ubuntu:16.04
Cache folder set to /home/.../.singularity/docker
[5/5] |===================================| 100.0%
Importing: base Singularity environment
Importing: /home/.../.singularity/docker/
↪sha256:1be7f2b886e89a58e59c4e685fcc5905a26ddef3201f290b96f1eff7d778e122.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:6fbc4a21b806838b63b774b338c6ad19d696a9e655f50b4e358cc4006c3baa79.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:c71a6f8e13782fed125f2247931c3eb20cc0e6428a5d79edb546f1f1405f0e49.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:4be3072e5a37392e32f632bb234c0b461ff5675ab7e362afad6359fbd36884af.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:06c6d2f5970057aef3aef6da60f0fde280db1c077f0cd88ca33ec1a70a9c7b58.tar.gz
Importing: /home/.../.singularity/metadata/
↪sha256:c6a9ef4b9995d615851d7786fbc2fe72f72321bee1a87d66919b881a0336525a.tar.gz
WARNING: Building container as an unprivileged user. If you run this container as root
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ./ubuntu-16.04.simg
Cleaning up...
Done. Container is at: ./ubuntu-16.04.simg
```

Note, there are some Warning messages concerning the build from Docker.

The example below does the same as above, but renames the image.

```
$ singularity pull --name ubuntu_docker.simg docker://ubuntu
Importing: /home/***/.singularity/docker/
↪sha256:c71a6f8e13782fed125f2247931c3eb20cc0e6428a5d79edb546f1f1405f0e49.tar.gz
Importing: /home/***/.singularity/docker/
↪sha256:4be3072e5a37392e32f632bb234c0b461ff5675ab7e362afad6359fbd36884af.tar.gz
Importing: /home/***/.singularity/docker/
↪sha256:06c6d2f5970057aef3aef6da60f0fde280db1c077f0cd88ca33ec1a70a9c7b58.tar.gz
Importing: /home/***/.singularity/metadata/
↪sha256:c6a9ef4b9995d615851d7786fbc2fe72f72321bee1a87d66919b881a0336525a.tar.gz
WARNING: Building container as an unprivileged user. If you run this container as root
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ./ubuntu_docker.simg
```

(continues on next page)

(continued from previous page)

```
Cleaning up...
Done. Container is at: ./ubuntu_docker.simg
```

When we run this particular Docker container without any runtime arguments, it does not return any notifications, and the Bash prompt does not change the prompt.

```
$ singularity run ubuntu_docker.simg
$ cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.3 LTS"
NAME="Ubuntu"
VERSION="16.04.3 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.3 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

Whoa, we're inside a container!?!

This is the OS on the VM I tested this on:

```
$ exit
exit
$ cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.1 LTS"
NAME="Ubuntu"
VERSION="16.04.1 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.1 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

Here we are back in the container:

```
$ singularity shell ubuntu_docker.simg
Singularity: Invoking an interactive shell within container...

Singularity ubuntu_docker.simg:~> cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.3 LTS"
```

(continues on next page)

(continued from previous page)

```
NAME="Ubuntu"
VERSION="16.04.3 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.3 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
Singularity ubuntu_docker.simg:~>
```

When invoking a container, make sure it executes and exits, or notifies you it is running.

Keeping track of downloaded images may be necessary if space is a concern.

By default, Singularity uses a temporary cache to hold Docker tarballs:

```
$ ls ~/.singularity
```

You can change these by specifying the location of the cache and temporary directory on your localhost:

```
$ sudo mkdir tmp
$ sudo mkdir scratch

$ SINGULARITY_TMPDIR=$PWD/scratch SINGULARITY_CACHEDIR=$PWD/tmp singularity --debug␣
↪pull --name ubuntu-tmpdir.simg docker://ubuntu
```

As an example, using Singularity we can run a UI program that was built from Docker, here I show the IDE RStudio *tidyverse* from Rocker

```
$ singularity exec docker://rocker/tidyverse:latest R
```

"An Introduction to Rocker: Docker Containers for R by Carl Boettiger, Dirk Eddelbuettel"

12.4 4. Building Singularity containers locally

Like Docker which uses a *dockerfile* to build its containers, Singularity uses a file called *Singularity*

When you are building locally, you can name this file whatever you wish, but a better practice is to put it in a directory and name it *Singularity* - as this will help later on when developing on Singularity-Hub and Github.

Create Container and add content to it:

```
$ singularity image.create ubuntu14.simg
Creating empty 768MiB image file: ubuntu14.simg
Formatting image with ext3 file system
Image is done: ubuntu14.simg

$ singularity build ubuntu14.simg docker://ubuntu:14.04
Building into existing container: ubuntu14.simg
Docker image path: index.docker.io/library/ubuntu:14.04
Cache folder set to /home/.../.singularity/docker
[5/5] |===================================| 100.0%
```

(continues on next page)

(continued from previous page)

```
Importing: base Singularity environment
Importing: /home/.../.singularity/docker/
↪sha256:c954d15f947c57e059f67a156ff2e4c36f4f3e59b37467ff865214a88ebc54d6.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:c3688624ef2b94ab3981564e23e1f48df8f1b988519373ccfb79d7974017cb85.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:848fe4263b3b44987f0eacdb2fc0469ae6ff04b2311e759985dfd27ae5d3641d.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:23b4459d3b04aa0bc7cb7f7021e4d7bbb5e87aa74a6a5f57475a0e8badbd9a26.tar.gz
Importing: /home/.../.singularity/docker/
↪sha256:36ab3b56c8f1a3188464886cbe41f42a969e6f9374e040f13803d796ed27b0ec.tar.gz
Importing: /home/.../.singularity/metadata/
↪sha256:c6a9ef4b9995d615851d7786fbc2fe72f72321bee1a87d66919b881a0336525a.tar.gz
WARNING: Building container as an unprivileged user. If you run this container as root
WARNING: it may be missing some functionality.
Building Singularity image...
Singularity container built: ubuntu14.simg
Cleaning up...
```

Note, *image.create* uses an ext3 file system

Create a container using a custom Singularity file:

```
$ singularity build --name ubuntu.simg Singularity
```

In the above command:

- *–name* will create a container named *ubuntu.simg*

Pull a Container from Docker and make it writable using the *–writable* flag:

```
$ sudo singularity build --writable ubuntu.simg  docker://ubuntu

Docker image path: index.docker.io/library/ubuntu:latest
Cache folder set to /root/.singularity/docker
Importing: base Singularity environment
Importing: /root/.singularity/docker/
↪sha256:1be7f2b886e89a58e59c4e685fcc5905a26ddef3201f290b96f1eff7d778e122.tar.gz
Importing: /root/.singularity/docker/
↪sha256:6fbc4a21b806838b63b774b338c6ad19d696a9e655f50b4e358cc4006c3baa79.tar.gz
Importing: /root/.singularity/docker/
↪sha256:c71a6f8e13782fed125f2247931c3eb20cc0e6428a5d79edb546f1f1405f0e49.tar.gz
Importing: /root/.singularity/docker/
↪sha256:4be3072e5a37392e32f632bb234c0b461ff5675ab7e362afad6359fbd36884af.tar.gz
Importing: /root/.singularity/docker/
↪sha256:06c6d2f5970057aef3aef6da60f0fde280db1c077f0cd88ca33ec1a70a9c7b58.tar.gz
Importing: /root/.singularity/metadata/
↪sha256:c6a9ef4b9995d615851d7786fbc2fe72f72321bee1a87d66919b881a0336525a.tar.gz
Creating empty Singularity writable container 120MB
Creating empty 150MiB image file: ubuntu.simg
Formatting image with ext3 file system
Image is done: ubuntu.simg
Building Singularity image...
Singularity container built: ubuntu.simg
Cleaning up...

$ singularity shell ubuntu.simg
```

(continues on next page)

(continued from previous page)

```
Singularity: Invoking an interactive shell within container...

Singularity ubuntu.simg:~> apt-get update

Reading package lists... Done
W: chmod 0700 of directory /var/lib/apt/lists/partial failed -␣
↪SetupAPTPartialDirectory (1: Operation not permitted)
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
E: Unable to lock directory /var/lib/apt/lists/
Singularity ubuntu.simg:~> exit
exit

$ sudo singularity shell ubuntu.simg

Singularity: Invoking an interactive shell within container...

Singularity ubuntu.simg:~> apt-get update

Hit:1 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:5 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [73.2 kB]
Get:6 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:7 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [585 kB]
Get:8 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [405␣
↪kB]
Get:9 http://security.ubuntu.com/ubuntu xenial-security/multiverse amd64 Packages␣
↪[3486 B]
Get:10 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
Get:11 http://archive.ubuntu.com/ubuntu xenial/multiverse amd64 Packages [176 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial-updates/universe Sources [241 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [953 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial-updates/restricted amd64 Packages [13.
↪1 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [762␣
↪kB]
Get:16 http://archive.ubuntu.com/ubuntu xenial-updates/multiverse amd64 Packages [18.
↪5 kB]
Get:17 http://archive.ubuntu.com/ubuntu xenial-backports/main amd64 Packages [5153 B]
Get:18 http://archive.ubuntu.com/ubuntu xenial-backports/universe amd64 Packages␣
↪[7168 B]
Fetched 23.2 MB in 4s (5569 kB/s)
Reading package lists... Done

Singularity ubuntu.simg:~> apt-get install curl --fix-missing
```

When I try to install software to the image without *sudo* it is denied, because root is the owner of the container. When I use *sudo* I can install software to the container. The software remain in the container after closing the container and restart.

Note: Bootstrapping *bootstrap* command is deprecated (v2.4), use *build* instead.

To install a container with Ubuntu from the ubuntu.com reposutiry you need to use *debootstrap*

12.4.1 Exercise 3: Creating the Singularity file (30 minutes)

Recipes can use any number of container registries for bootstrapping a container.

(Advanced) the *Singularity* file can be hosted on Github and will be auto-detected by Singularity-Hub when you set up your Container Collection.

Building your own containers requires that you have *sudo* privileges - therefore you'll need to develop these on your local machine or on a VM that you can gain root access on.

- The Header

The top of the file, selects the base OS for the container. *Bootstrap:* references the repository (e.g. *docker*, *debootstrap*, *sub*). *From:* selects the name of the owner/container.

```
Bootstrap: shub
From: vsoch/hello-world
```

Using *debootstrap* with a build that uses a mirror:

```
BootStrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/
```

Using a *localimage* to build:

```
Bootstrap: localimage
From: /path/to/container/file/or/directory
```

Using CentOS-like container:

```
Bootstrap: yum
OSVersion: 7
MirrorURL: http://mirror.centos.org/centos-7/7/os/x86_64/
Include:yum
```

Note: to use *yum* to build a container you should be operating on a RHEL system, or an Ubuntu system with *yum* installed.

The container registries which Singularity uses are listed above in Section 3.1.

- Sections

The Singularity file uses sections to specify the dependencies, environmental settings, and runscripts when it build.

- %help - create text for a help menu associated with your container
- %setup - executed on the host system outside of the container, after the base OS has been installed.
- %files - copy files from your host system into the container
- %labels - store metadata in the container
- %environment - loads environment variables at the time the container is run (not built)
- %post - set environment variables during the build
- %runscript - executes a script when the container runs
- %test - runs a test on the build of the container
- Apps

In Singularity 2.4+ we can build a container which does multiple things, e.g. each app has its own runscripts. These use the prefix *%app* before the sections mentioned above. The *%app* architecture can exist in addition to the regular *%post* and *%runscript* sections.

```
Bootstrap: docker
From: ubuntu

% environment

%labels

##############################
# foo
##############################

%apprun foo
    exec echo "RUNNING FOO"

%applabels foo
    BESTAPP=FOO
    export BESTAPP

%appinstall foo
    touch foo.exec

%appenv foo
    SOFTWARE=foo
    export SOFTWARE

%apphelp foo
    This is the help for foo.

%appfiles foo
    avocados.txt


##############################
# bar
##############################

%apphelp bar
    This is the help for bar.

%applabels bar
    BESTAPP=BAR
    export BESTAPP

%appinstall bar
    touch bar.exec

%appenv bar
    SOFTWARE=bar
    export SOFTWARE
```

- Setting up Singularity file system

%help section can be as verbose as you want

```
Bootstrap: docker
From: ubuntu

%help
This is the container help section.
```

%setup commands are executed on the localhost system outside of the container - these files could include necessary build dependencies. We can copy files to the *$SINGULARITY_ROOTFS* file system can be done during *%setup*

%files include any files that you want to copy from your localhost into the container.

%post includes all of the environment variables and dependencies that you want to see installed into the container at build time.

%environment includes the environment variables which we want to be run when we start the container

%runscript does what it says, it executes a set of commands when the container is run.

Example Singularity file bootstrapping a Docker Ubuntu (16.04) image.

```
BootStrap: docker
From: ubuntu:16.04

%post
    apt-get -y update
    apt-get -y install fortune cowsay lolcat

%environment
    export LC_ALL=C
    export PATH=/usr/games:$PATH

%runscript
    fortune | cowsay | lolcat

%labels
    Maintainer Tyson Swetnam
    Version v0.1
```

Build the container:

```
singularity build --name cowsay_container.simg Singularity
```

Run the container:

```
singularity run cowsay.simg
```

If you build a *squashfs* container, it is immutable (you cannot –*writable* edit it)

12.5 5. Running Singularity Containers

Commands:

exec - command allows you to execute a custom command within a container by specifying the image file.

shell - command allows you to spawn a new shell within your container and interact with it.

run - assumes your container is set up with "runscripts" triggered with the *run* command, or simply by calling the container as though it were an executable.

inspect - inspects the container.

–*writable* - creates a writable container that you can edit interactively and save on exit.

–*sandbox* - copies the guts of the container into a directory structure.

12.5.1 5.1 Using the *exec* command

```
$ singularity exec shub://singularityhub/ubuntu cat /etc/os-release
```

12.5.2 5.2 Using the *shell* command

```
$ singularity shell shub://singularityhub/ubuntu
```

12.5.3 5.3 Using the *run* command

```
$ singularity run shub://singularityhub/ubuntu
```

12.5.4 5.4 Using the *inspect* command

You can inspect the build of your container using the *inspect* command

```
$ singularity pull  shub://vsoch/hello-world
Progress |===================================| 100.0%
Done. Container is at: /home/***/vsoch-hello-world-master-latest.simg

$ singularity inspect vsoch-hello-world-master-latest.simg
{
    "org.label-schema.usage.singularity.deffile.bootstrap": "docker",
    "MAINTAINER": "vanessasaur",
    "org.label-schema.usage.singularity.deffile": "Singularity",
    "org.label-schema.schema-version": "1.0",
    "WHATAMI": "dinosaur",
    "org.label-schema.usage.singularity.deffile.from": "ubuntu:14.04",
    "org.label-schema.build-date": "2017-10-15T12:52:56+00:00",
    "org.label-schema.usage.singularity.version": "2.4-feature-squashbuild-secbuild.
↪g780c84d",
    "org.label-schema.build-size": "333MB"
}
```

12.5.5 5.5 Using the –*sandbox* and –*writable* commands

As of Singularity v2.4 by default *build* produces immutable images in the 'squashfs' file format. This ensures reproducible and verifiable images.

Creating a –*writable* image must use the *sudo* command, thus the owner of the container is *root*

```
$ sudo singularity build --writable ubuntu-master.simg shub://singularityhub/ubuntu
Cache folder set to /root/.singularity/shub
Progress |===================================| 100.0%
Building from local image: /root/.singularity/shub/singularityhub-ubuntu-master-
↪latest.simg
Creating empty Singularity writable container 208MB
Creating empty 260MiB image file: ubuntu-master.simg
Formatting image with ext3 file system
Image is done: ubuntu-master.simg
Building Singularity image...
Singularity container built: ubuntu-master.simg
Cleaning up...
```

You can convert these images to writable versions using the *–writable* and *–sandbox* commands.

When you use the *–sandbox* the container is written into a directory structure. Sandbox folders can be created without the *sudo* command.

```
$ singularity build --sandbox lolcow/ shub://GodloveD/lolcow
WARNING: Building sandbox as non-root may result in wrong file permissions
Cache folder set to /home/.../.singularity/shub
Progress |===================================| 100.0%
Building from local image: /home/.../.singularity/shub/GodloveD-lolcow-master-latest.
↪simg
WARNING: Building container as an unprivileged user. If you run this container as root
WARNING: it may be missing some functionality.
Singularity container built: lolcow/
Cleaning up...
@vm142-73:~$ cd lolcow/
@vm142-73:~/lolcow$ ls
bin  boot  dev  environment  etc  home  lib  lib64  media  mnt  opt  proc  run  sbin ␣
↪singularity  srv  sys  tmp  usr  var
```

12.5.6 5.6 Test

Singularity can test the build of your container.

You can bypass the test by using *–no-test*.

12.5.7 5.7 Bind Paths

When Singularity creates the new file system inside a container it ignores directories that are not part of the standard kernel, e.g. */scratch*, */xdisk*, */global*, etc. These paths can be added back into the container by binding them when the container is run.

```
$ singularity shell --bind /xdisk ubuntu14.simg
```

The system administrator can also define what is added to a container. This is important on campus HPC systems that often have a */scratch* or */xdisk* directory structure. By editing the */etc/singularity/singularity.conf* a new path can be added to the system containers.

12.5.8 5.8 Overlay

You can make changes to an immutable container which only persist for the duration of the container being run.

First, download a container.

Next, create a new image in the ext3 format.

```
$ singularity image.create blank_slate.simg
```

Now, overlay your blank image file name with the container you just downloaded.

```
$ sudo singularity shell --overlay blank_slate.simg ubuntu14.simg
```

note: using the ‘sudo‘ command to make the container writable

CHAPTER 13

Advanced Singularity

13.1 1. Using HPC Environments

Conducting analyses on high performance computing clusters happens through very different patterns of interaction than running analyses on a VM. When you login, you are on a node that is shared with lots of people. Trying to run jobs on that node is not "high performance" at all. Those login nodes are just intended to be used for moving files, editing files, and launching jobs.

Most jobs on an HPC cluster are neither interactive, nor realtime. When you submit a job to the scheduler, you must tell it what resources you need (e.g. how many nodes, what type of nodes) and what you want to run. Then the scheduler finds resources matching your requirements, and runs the job for you when it can.

For example, if you want to run the command:

```
singularity exec docker://python:latest /usr/local/bin/python
```

On an HPC system, your job submission script would look something like:

```
#!/bin/bash
#
#SBATCH -J myjob                    # Job name
#SBATCH -o output.%j                # Name of stdout output file (%j expands to␣
↪jobId)
#SBATCH -p development              # Queue name
#SBATCH -N 1                        # Total number of nodes requested (68 cores/
↪node)
#SBATCH -n 17                       # Total number of mpi tasks requested
#SBATCH -t 02:00:00                 # Run time (hh:mm:ss) - 4 hours

module load tacc-singularity
singularity exec docker://python:latest /usr/local/bin/python
```

This example is for the Slurm scheduler, a popular one used by all TACC systems. Each of the #SBATCH lines looks like a comment to the bash kernel, but the scheduler reads all those lines to know what resources to reserve for you.

It is usually possible to get an interactive session as well. At TACC, the command "idev" is used to get an interactive development session. For example:

```
idev -m 240 -p skx-normal
```

Just running "idev" gives you one hour on a development node. The example above will give you a 240 minute interactive session on the "skx-normal" partition of the system (which in this case have 48 Skylake CPU cores per node).

Note: Every HPC cluster is a little different, but they almost universally have a "User's Guide" that serves both as a quick reference for helpful commands and contains guidelines for how to be a "good citizen" while using the system. For TACC's Stampede2 system, the user guide is at: https://portal.tacc.utexas.edu/user-guides/stampede2

13.1.1 How do HPC systems fit into the development workflow?

A few things to consider when using HPC systems:

1. Using 'sudo' is not allowed on HPC systems, and building a Singularity container from scratch requires sudo. That means you have to build your containers on a different development system. You can pull a docker image on HPC systems
2. If you need to edit text files, command line text editors don't support using a mouse, so working efficiently has a learning curve. There are text editors that support editing files over SSH. This lets you use a local text editor and just save the changes to the HPC system.
3. Singularity is in the process of changing image formats. Depending on the version of Singularity running on the HPC system, new squashFS or .simg formats may not work.

13.2 2. Singularity and MPI

Singularity supports MPI fairly well. Since (by default) the network is the same insde and outside the container, the communication between containers usually just works. The more complicated bit is making sure that the container has the right set of MPI libraries. MPI is an open specification, but there are several implementations (OpenMPI, MVAPICH2, and Intel MPI to name three) with some non-overlapping feature sets. If the host and container are running different MPI implementations, or even different versions of the same implementation, hilarity may ensue.

The general rule is that you want the version of MPI inside the container to be the same version or newer than the host. You may be thinking that this is not good for the portability of your container, and you are right. Containerizing MPI applications is not terribly difficult with Singularity, but it comes at the cost of additional requirements for the host system.

Note: Many HPC Systems, like Stampede2, have highspeed, low latency networks that have special drivers. Infiniband, Ares, and OmniPath are three different specs for these types of networks. When running MPI jobs, if the container doesn't have the right libraries, it won't be able to use those special interconnects to communicate between nodes.

Because you may have to build your own MPI enabled Singularity images (to get the versions to match), here is a 2.3 compatible example of what it may look like:

```
# Copyright (c) 2015-2016, Gregory M. Kurtzer. All rights reserved.
#
# "Singularity" Copyright (c) 2016, The Regents of the University of     California,
# through Lawrence Berkeley National Laboratory (subject to receipt of any
# required approvals from the U.S. Dept. of Energy).  All rights reserved.

BootStrap: debootstrap
OSVersion: xenial
MirrorURL: http://us.archive.ubuntu.com/ubuntu/


%runscript
    echo "This is what happens when you run the container..."


%post
    echo "Hello from inside the container"
    sed -i 's/$/ universe/' /etc/apt/sources.list
    apt update
    apt -y --allow-unauthenticated install vim build-essential wget     gfortran␣
↪bison libibverbs-dev libibmad-dev libibumad-dev librdmacm-dev     libmlx5-dev␣
↪libmlx4-dev
    wget http://mvapich.cse.ohio-state.edu/download/mvapich/mv2/    mvapich2-2.1.tar.
↪gz
    tar xvf mvapich2-2.1.tar.gz
    cd mvapich2-2.1
    ./configure --prefix=/usr/local
    make -j4
    make install
    /usr/local/bin/mpicc examples/hellow.c -o /usr/bin/hellow
```

You could also build in everything in a Dockerfile and convert the image to Singularity at the end.

Once you have a working MPI container, invoking it would look something like:

```
mpirun -np 4 singularity exec ./mycontainer.img /app.py arg1 arg2
```

This will use the **host MPI** libraries to run in parallel, and assuming the image has what it needs, can work across many nodes.

For a single node, you can also use the **container MPI** to run in parallel (usually you don't want this)

```
singularity exec ./mycontainer.img mpirun -np 4 /app.py arg1 arg2
```

13.3 3. Singularity and GPU Computing

GPU support in Singularity is fantastic

Since Singularity supported docker containers, it has been fairly simple to utilize GPUs for machine learning code like TensorFlow. From Maverick, which is TACC's GPU system:

```
# Work from a compute node
idev -m 60
# Load the singularity module
module load tacc-singularity
# Pull your image
singularity pull docker://nvidia/caffe:latest

singularity exec --nv caffe-latest.img caffe device_query -gpu 0
```

Please note that the –nv flag specifically passes the GPU drivers into the container. If you leave it out, the GPU will not be detected.

```
singularity exec caffe-latest.img caffe device_query -gpu 0
```

For TensorFlow, you can directly pull their latest GPU image and utilize it as follows.

```
# Change to your $WORK directory
cd $WORK
#Get the software
git clone https://github.com/tensorflow/models.git ~/models
# Pull the image
singularity pull docker://tensorflow/tensorflow:latest-gpu
# Run the code
singularity exec --nv tensorflow-latest-gpu.img python $HOME/models/tutorials/image/
↪mnist/convolutional.py
```

Note: You probably noticed that we check out the models repository into your $HOME directory. This is because your $HOME and $WORK directories are only available inside the container if the root folders /home and /work exist inside the container. In the case of tensorflow-latest-gpu.img, the /work directory does not exist, so any files there are inaccessible to the container.

You may be thinking "what about overlayFS??". Stampede2 supports it, but the Linux kernel on the other systems does not support overlayFS, so it had to be disabled in our Singularity install. This may change as new Singularity versions are released.

13.3.1 Hands-On Exercise

Build a Singularity container that implements a simple Tensorflow image classifier.

The image classifier script is available "out of the box" here: https://raw.githubusercontent.com/tensorflow/models/master/tutorials/image/imagenet/classify_image.py

Tensorflow has working Docker containers on DockerHub that you can use to support all the dependencies. For example, the first line of your Dockerfile might look like:

```
FROM tensorflow/tensorflow:1.5.0-py3
```

When running the image classifier, the non-containerized version would be invoked with something like:

```
python /classify_image.py --model_dir /model --image_file cat.png
```

You can use a Singularity file or a Dockerfile to help you. For reference, you can lookback at the "Singularity Intro" section on building Singularity images, yesterday's material on building Dockerfiles, or the respective manual pages:

- http://singularity.lbl.gov/docs-build-container
- https://docs.docker.com/engine/reference/builder/

CHAPTER 14

Deploying apps in Discovery Environment

The CyVerse Discovery Environment (DE) provides a simple yet powerful web portal for managing data, analyses, and workflows. The DE uses containers (via Docker and Singularity through Agave) to support customizable, non-interactive, reproducible workflows using data stored in the CyVerse Data Store, based on iRODS. Agave is a "Science-as-a-Service" API platform for high-performance computing (HPC), high-throughput computing (HTC) and big-data resources.

14.1 Deploying Docker images as apps in DE

Instruction guide: This paper will guide you to bring your dockerized tools into CyVerse DE.

https://f1000research.com/articles/5-1442/v3

F1000Research

F1000Research 2016, 5:1442 Last updated: 05 DEC 2016

CrossMark
click for updates

SOFTWARE TOOL ARTICLE

REVISED Bringing your tools to CyVerse Discovery Environment using Docker [version 3; referees: 3 approved]

Upendra Kumar Devisetty, Kathleen Kennedy, Paul Sarando, Nirav Merchant, Eric Lyons

CyVerse, University of Arizona, Tucson, AZ, 85721, USA

Note: Significant changes have been made as to how you can bring your tools into DE and so we are working on a separate paper that will show all those changes. Meanwhile you can follow the below tutorial for integrating your tools.

Here are the basic steps for deploying Docker images as apps in DE. For this tutorial I am going to show an example of Tensor image classifier docker image that I dockerized and pushed to dockerhub.

- *Build and test your Docker images*
- *Push your Docker image to public repositories*
- *Add Docker images as tool in DE*
- *Create a UI for the tool in DE*
- *Test the app using appropriate test data*

Warning: If you already have your own Docker image or Docker image of interest is hosted on public repositories (Dockerhub or quay.io or some other public repository), then you can skip to step 3

1. Build and test your Docker images

The first step is to dockerize your tool or software of interest. Detailed steps of how to dockerize your tool and test your dockerized images can be found in sections intro to docker and advanced docker.

For this tutorial I will use the `tensorflow image classifier` docker image that I built using this code

Testing

```
docker run -v $(pwd):/data -w /data tensorflow_up:1.0 sample_data/16401288243_
↪36112bd52f_m.jpg
```

2. Push your Docker image to public repositories

Once the Docker image works then you can push those images to some public repository such as dockerhub or quay.io

Here is the docker image for the tensorflow image classifier on docker hub - https://hub.docker.com/r/upendradevisetty/tensorflow_image_classifier/

3. Add Docker images as tool in DE

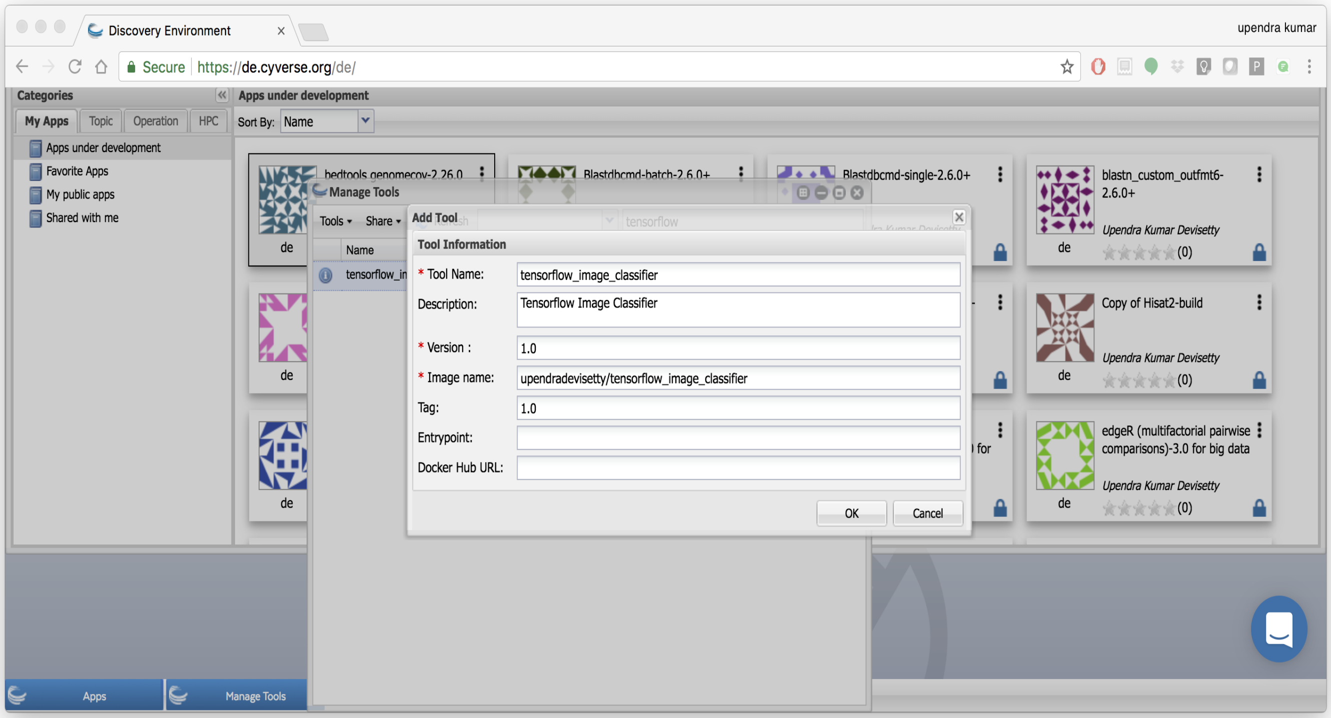
All tools now run installed as Docker images in the DE. Once the software is dockerized and available as Docker images on dockerhub then you can add those docker images as a tool in DE.

Warning: Check if the tool and correct version are already installed in the DE by following the below steps.

- Log in to the Discovery Environment by going to https://de.cyverse.org/de/, entering your CyVerse username and password, and clicking LOGIN. If you have not already done so, you will need to sign up for a CyVerse account.
- Click the `Apps` window to open the Apps window.
- Click the `Manage Tools` button on the top-right of the Apps window.
- In the search tools field, enter the first few letters of the tool name and then click enter.
- If the tool is available then you can skip to skip to step 3 for creating a UI for that tool.

If the tool is not available in DE then do the following:

- Click open the `Tools` tab in `Manage Tools` window and then click `Add tools` button
- Then enter the fields about your tool and then click "Ok".
  - Tool Name: It should be the name of the tool. For example "tensorflow_image_classifier".
  - Description: A short Description about the tool. For example "Tensorflow image classifier".
  - Version: What is the version number of the tool. For example "1.0".
  - Image name: Name of the Docker image on dockerhub or quay.io. For example "upendradevisetty/tensorflow_image_classifier".
  - Tag: What is the tag of your Docker image. This is optional but is highly recommended. If non specified, it will pull the default tag `latest`. If the `latest` tag is not avaiable the tool integration will fail. For example "1.0"
  - Entrypoint: Do you want a entrypoint for your Docker image? This optional.
  - Docker Hub URL: URL of the Dockerhub docker image. Option but is recommended. In this example "".



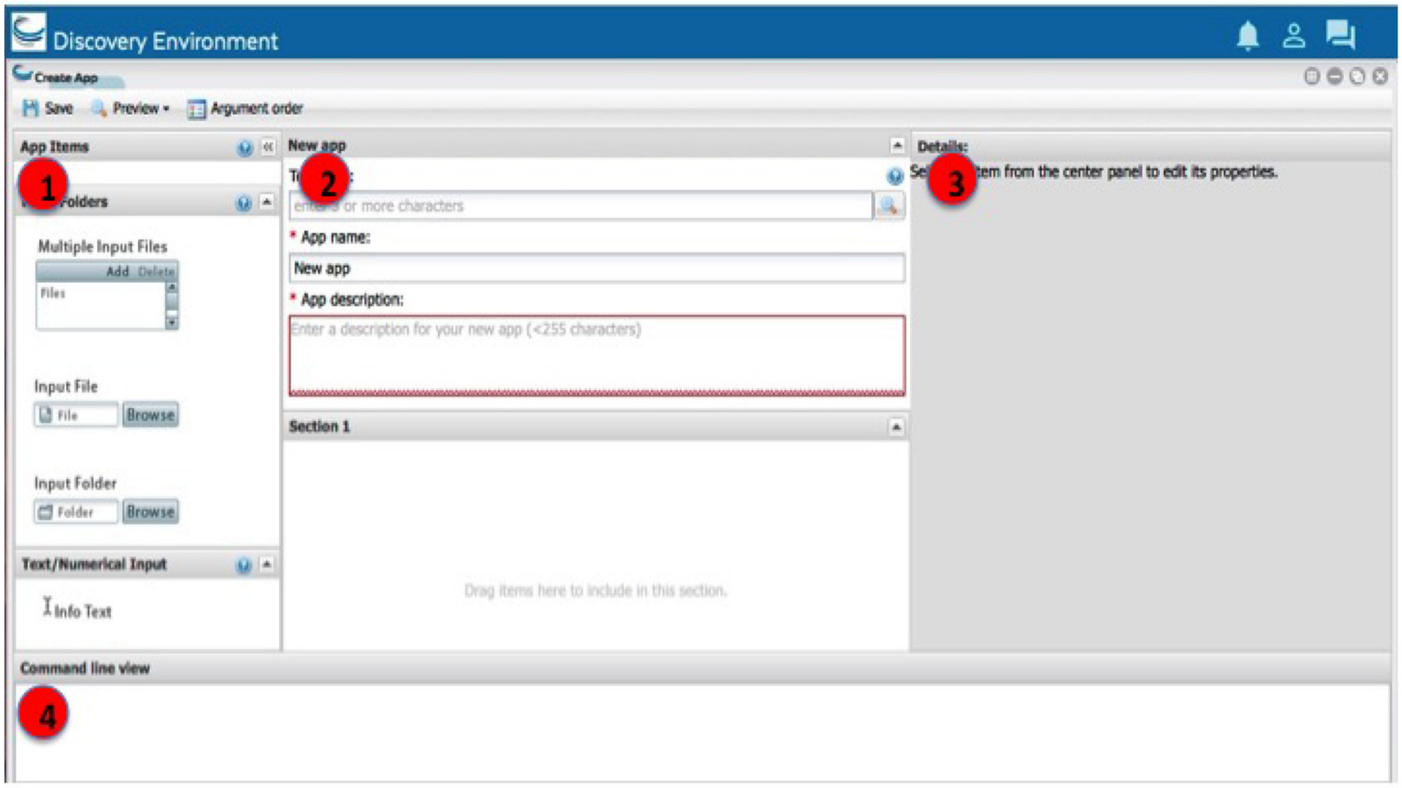
* Tool Name:	tensorflow_image_classifier
Description:	Tensorflow Image Classifier
* Version :	1.0
* Image name:	upendradevisetty/tensorflow_image_classifier
Tag:	1.0
Entrypoint:	
Docker Hub URL:	

- If there is no error, it indicates successful integration of the tool.

4. Create a UI for the tool in DE

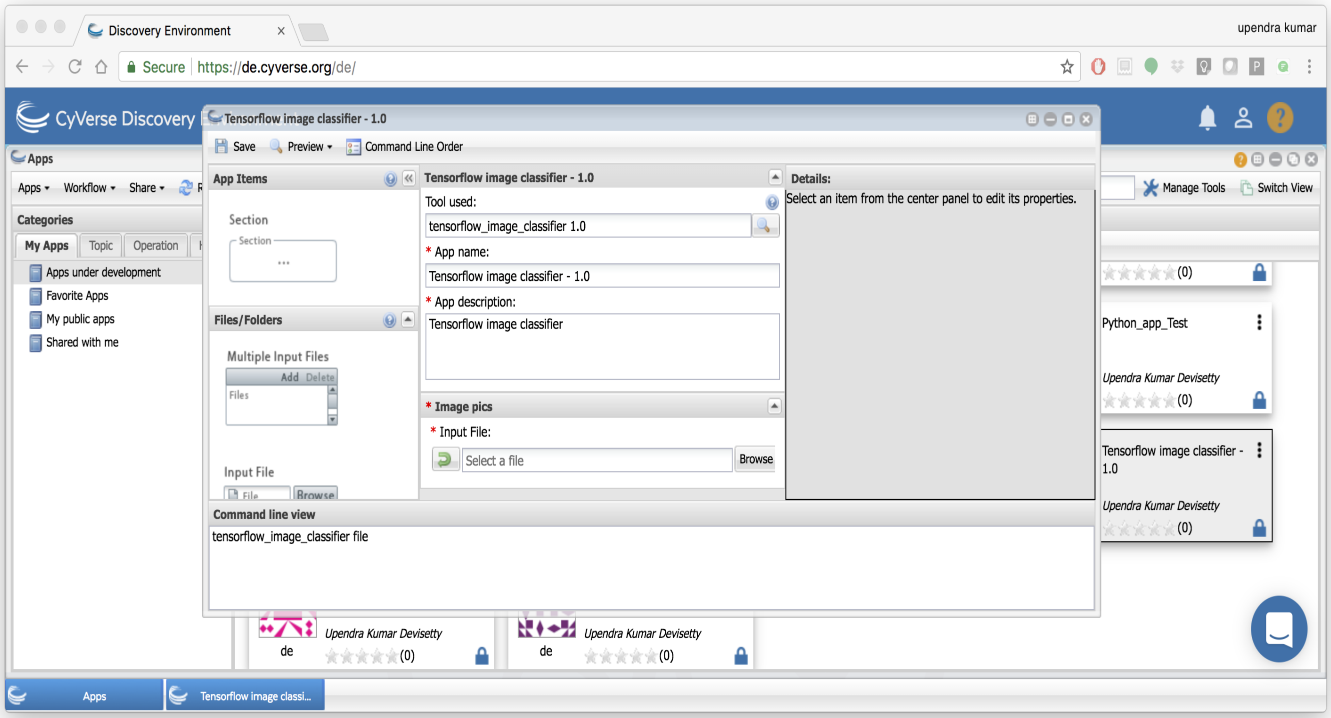
Once the Dockerized tool is added, you can create the app UI for the tool. The `Create App` window consists of four distinct sections:

- The first section contains the different app items that can be added to your interface. To add an app item, select the one to use (hover over the object name for a brief description) and drag it into position in the middle section.
- The second section is the landing place for the objects you dragged and dropped from the left section, and it updates to display how the app will look when presented to a user.
- The third section (Details) displays all of the available properties for the selected item. As you customize the app in this section, the middle section updates dynamically so you can see how it will look and act.
- Finally, the fourth section at the bottom (Command line view) contains the command-line commands for the current item's properties. As you update the properties in the Details section, the command-line view updates as well to let you make sure that you are passing the correct arguments in the correct order.



Note: Creating a new app interface requires that you know how to use the tool. With that knowledge, you create the interface according to how you want options to be displayed to a user.

Here is an example of the `Tensorflow image classifier - 1.0` app UI in DE



Tensorflow image classifier - 1.0

Tool used:
tensorflow_image_classifier 1.0

* App name:
Tensorflow image classifier - 1.0

* App description:
Tensorflow image classifier

* Image pics

* Input File:
Select a file

Select an item from the center panel to edit its properties.

Command line view
tensorflow_image_classifier file

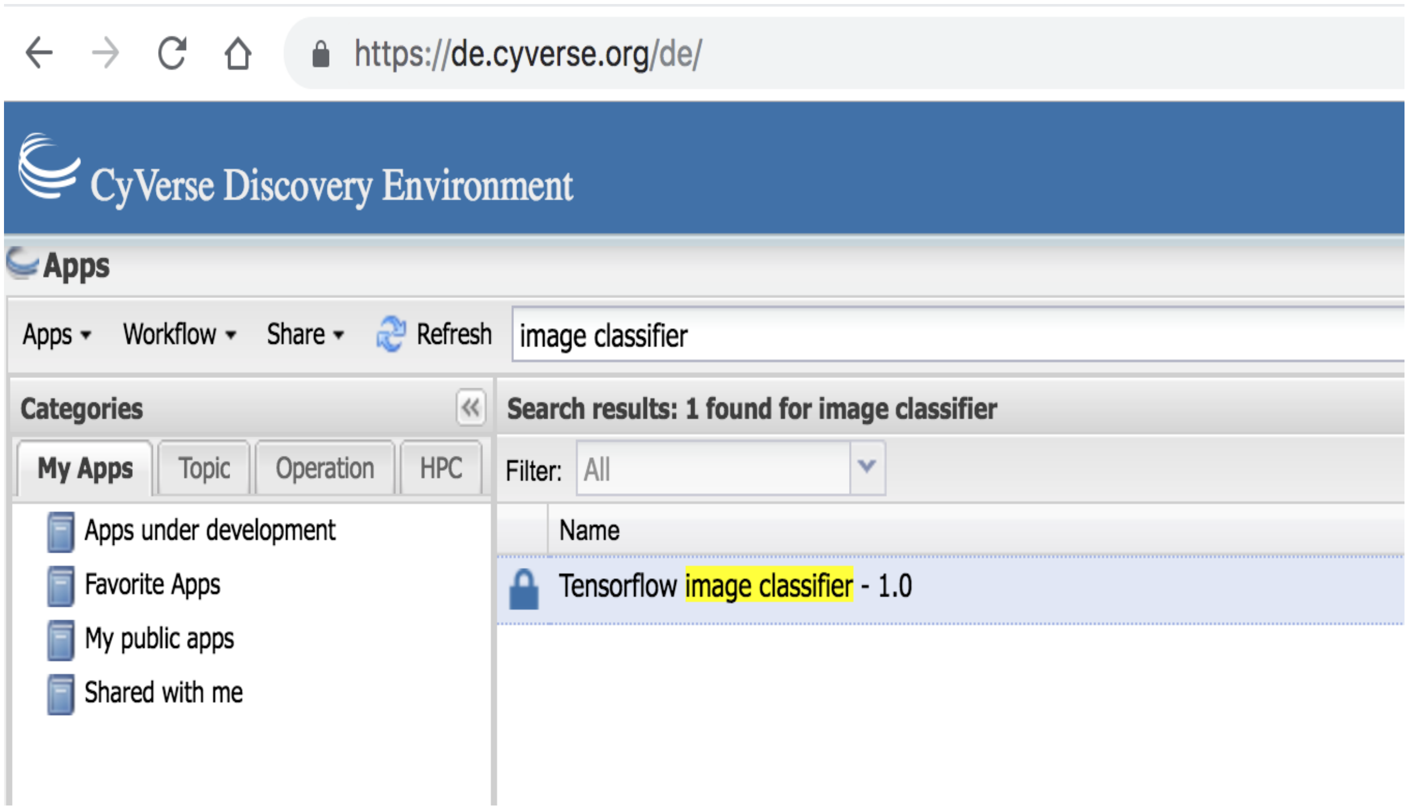
Python_app_Test
Upendra Kumar Devisetty (0)
Tensorflow image classifier - 1.0
Upendra Kumar Devisetty (0)
Upendra Kumar Devisetty de (0)
Upendra Kumar Devisetty de (0)

5. Test the app using appropriate test data

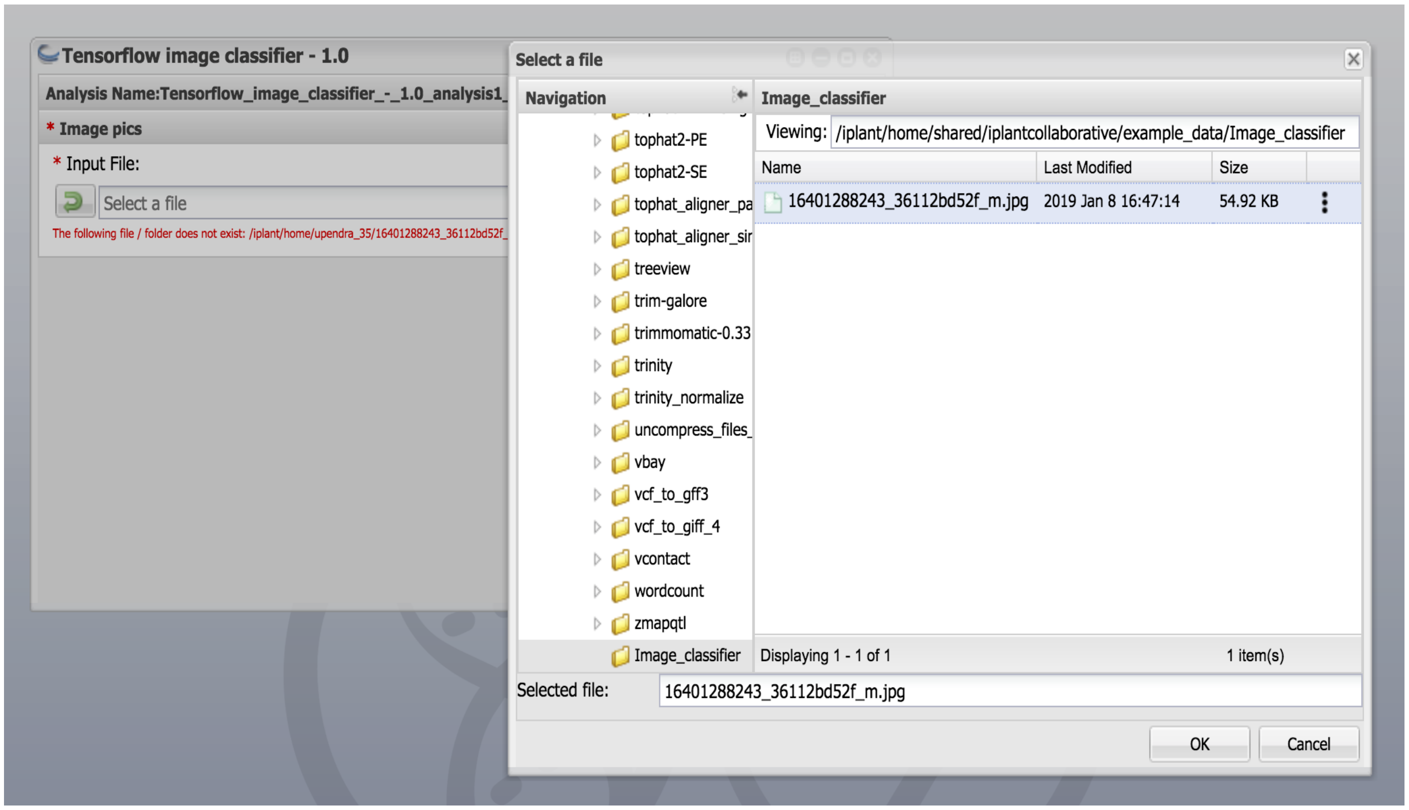
After creating the new app according to your design, test your app in the your Apps under development folder in the DE using appropriate test data to make sure it works properly.

For testing, we'll use the the same image that we have used earlier.

1. First open the `Tensorflow image classifier - 1.0` app in the app window



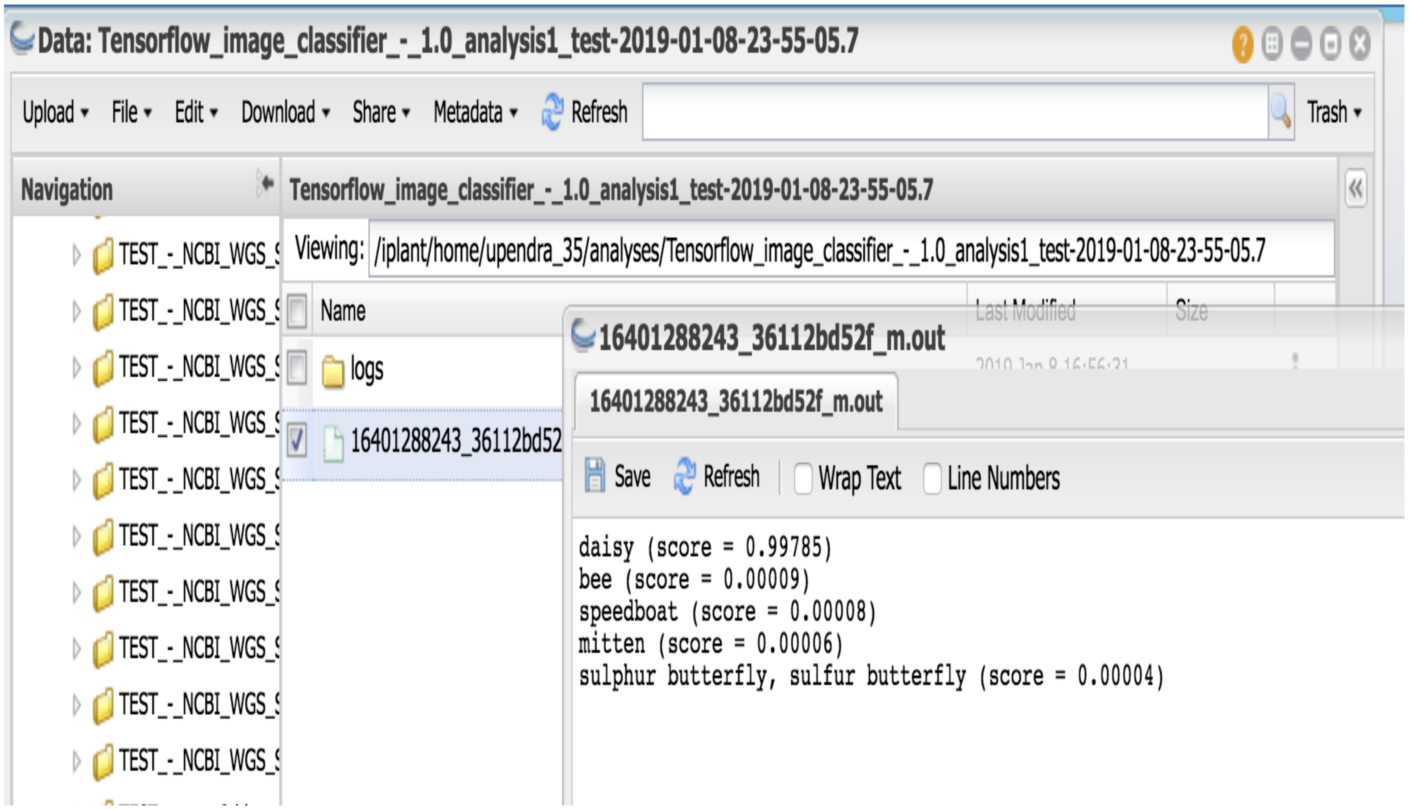
2. Next browse the test file in the app and click launch analysis



The following file / folder does not exist: /iplant/home/upendra_35/16401288243_36112bd52f_

Name	Last Modified	Size
16401288243_36112bd52f_m.jpg	2019 Jan 8 16:47:14	54.92 KB

3. After the analysis is completed, open the folder and check to see if the image classifier correctly predicts


Data: Tensorflow_image_classifier_-_1.0_analysis1_test-2019-01-08-23-55-05.7
Upload
File
Edit
Download
Share
Metadata
Refresh
Trash
Navigation
Tensorflow_image_classifier_-_1.0_analysis1_test-2019-01-08-23-55-05.7
Viewing: /iplant/home/upendra_35/analyses/Tensorflow_image_classifier_-_1.0_analysis1_test-2019-01-08-23-55-05.7
Name
Last Modified
Size
logs
16401288243_36112bd52
16401288243_36112bd52f_m.out
Save
Refresh
Wrap Text
Line Numbers

```
daisy (score = 0.99785)
bee (score = 0.00009)
speedboat (score = 0.00008)
mitten (score = 0.00006)
sulphur butterfly, sulfur butterfly (score = 0.00004)
```

Congrats!!! It works. The image classifier correctly predicts that image is a daisy..

- If your app works the way you expect it to you can share your app or make the app public
- If your app doesn't work, then you may need to make changes to the app UI or you need to make changes to your Docker image. If you make changes to the Docker image, then you don't need to create a new app UI again as the Docker image updates will be propagated automatically.

CHAPTER 15

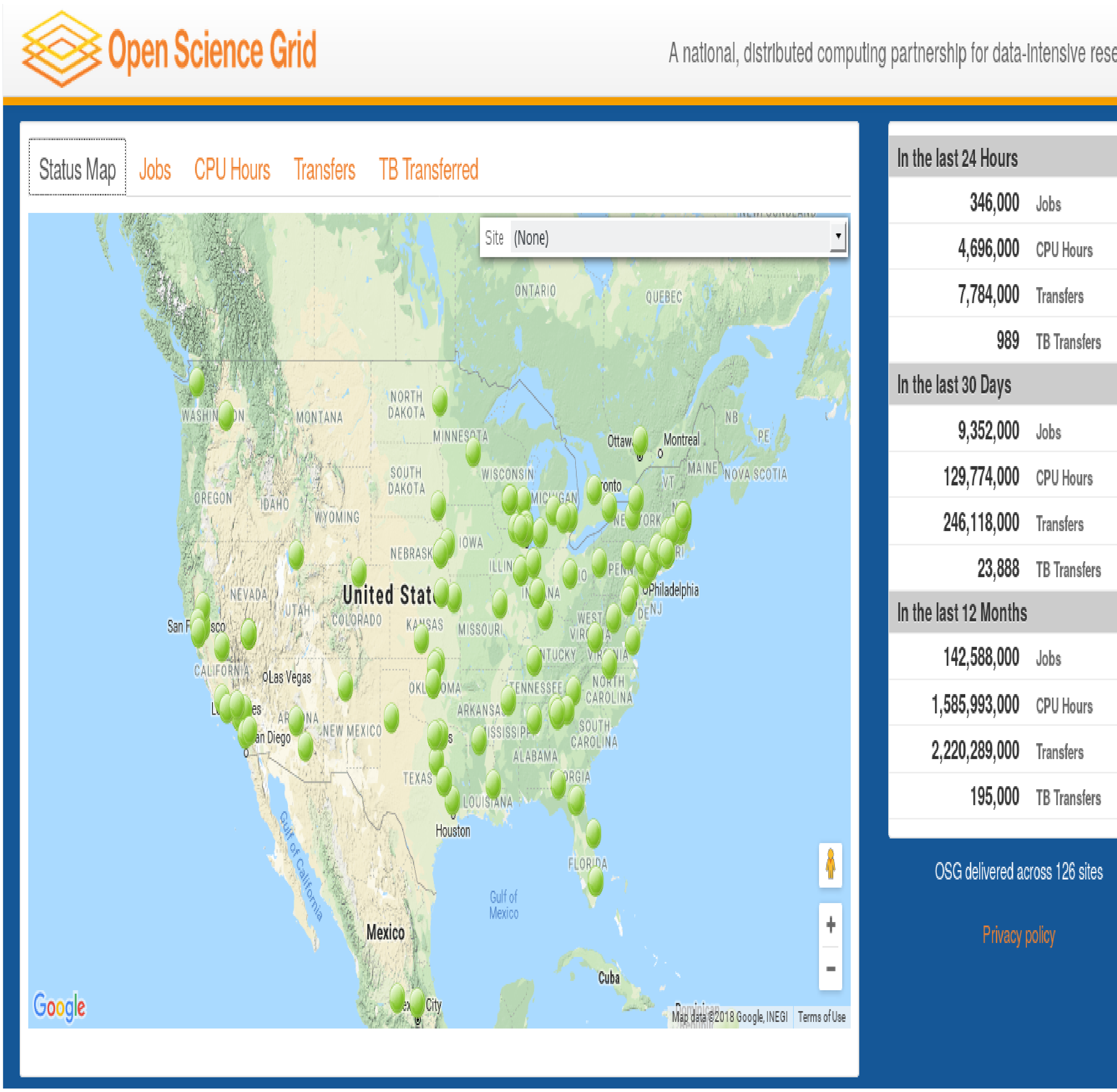
OSG (Open Science Grid) Singularity Infrastructure

15.1 1. Prerequisites

ssh will be used to connect to a remote job submit host. Please ensure you have a ssh client installed. The instructors will supply a slip of paper with username, password and hostname during the session.

15.2 2. OSG Overview

The Open Science Grid (OSG) is a distributed infrastructure for high throughput computing. The OSG enables dozens of universities and labs to provide researchers with access to significant computing resources. The resources accessible through the OSG are contributed by the community, organized by the OSG, and governed by the OSG consortium. In the last 12 months, we have provided more than 1.2 Billion CPU hours to researchers across a wide variety of projects.



In the last 24 Hours	
346,000	Jobs
4,696,000	CPU Hours
7,784,000	Transfers
989	TB Transfers
In the last 30 Days	
9,352,000	Jobs
129,774,000	CPU Hours
246,118,000	Transfers
23,888	TB Transfers
In the last 12 Months	
142,588,000	Jobs
1,585,993,000	CPU Hours
2,220,289,000	Transfers
195,000	TB Transfers

OSG delivered across 126 sites

Privacy policy

This map is available "live" on the OSG Display site.

15.2.1 2.1 Distributed High Throughput Computing

OSG is focusing on single core, or low number core jobs which can fit on a single node. Some guidelines:

- Use less than 2 GB memory
- Each invocation should run for 1-12 hours

- Compute sites in the OSG can be configured to use preemption, which means jobs can be automatically killed if higher priority jobs enter the system. Preempted jobs will restart on another site, but it is important that the jobs can handle multiple restarts.

15.2.2 2.2 Motivations for Containers in OSG

Each computing resource exposes a slightly different operating system environment. Actually this capacity can be daunting for the individual: each site has its own discretion to setup their runtime environment in a unique way.

Note: Before containers, OSG **used** to tell the users:

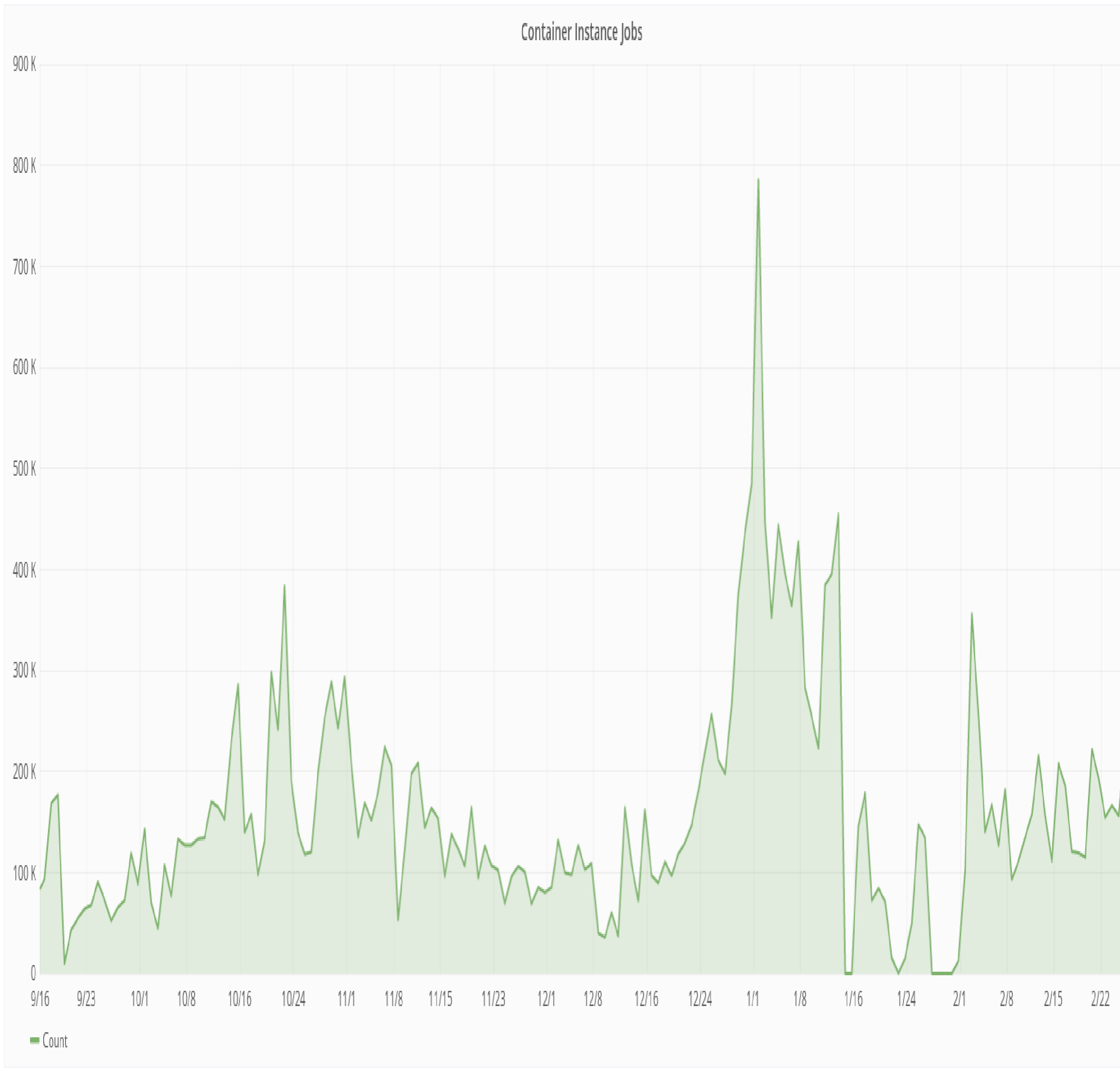
- Binaries should preferably be statically linked. However, dynamically linked binaries with standard library dependencies, built for a 64-bit Red Hat Enterprise Linux (RHEL) 6 machines will also work. Also, interpreted languages such as Python or Perl will work as long as there are no special module requirements.
- Software dependencies can be difficult to accommodate unless the software can be staged with the job.

Motivation for containers in OSG:

- **Consistent environment (default images)** - If a user does not specify a specific image, a default one is used by the job. The image contains a decent base line of software, and because the same image is used across all the sites, the user sees a more consistent environment than if the job landed in the environments provided by the individual sites.
- **Custom software environment (user defined images)** - Users can create and use their custom images, which is useful when having very specific software requirements or software stacks which can be tricky to bring with a job. For example: Python or R modules with dependencies, TensorFlow, …
- **Enables special environment such as GPUs** - Special software environments to go and in hand with the special hardware.
- **Process isolation** - Sandboxes the job environment so that a job can not peek at other jobs.
- **File isolation** - Sandboxes the job file system, so that a job can not peek at other jobs' data.

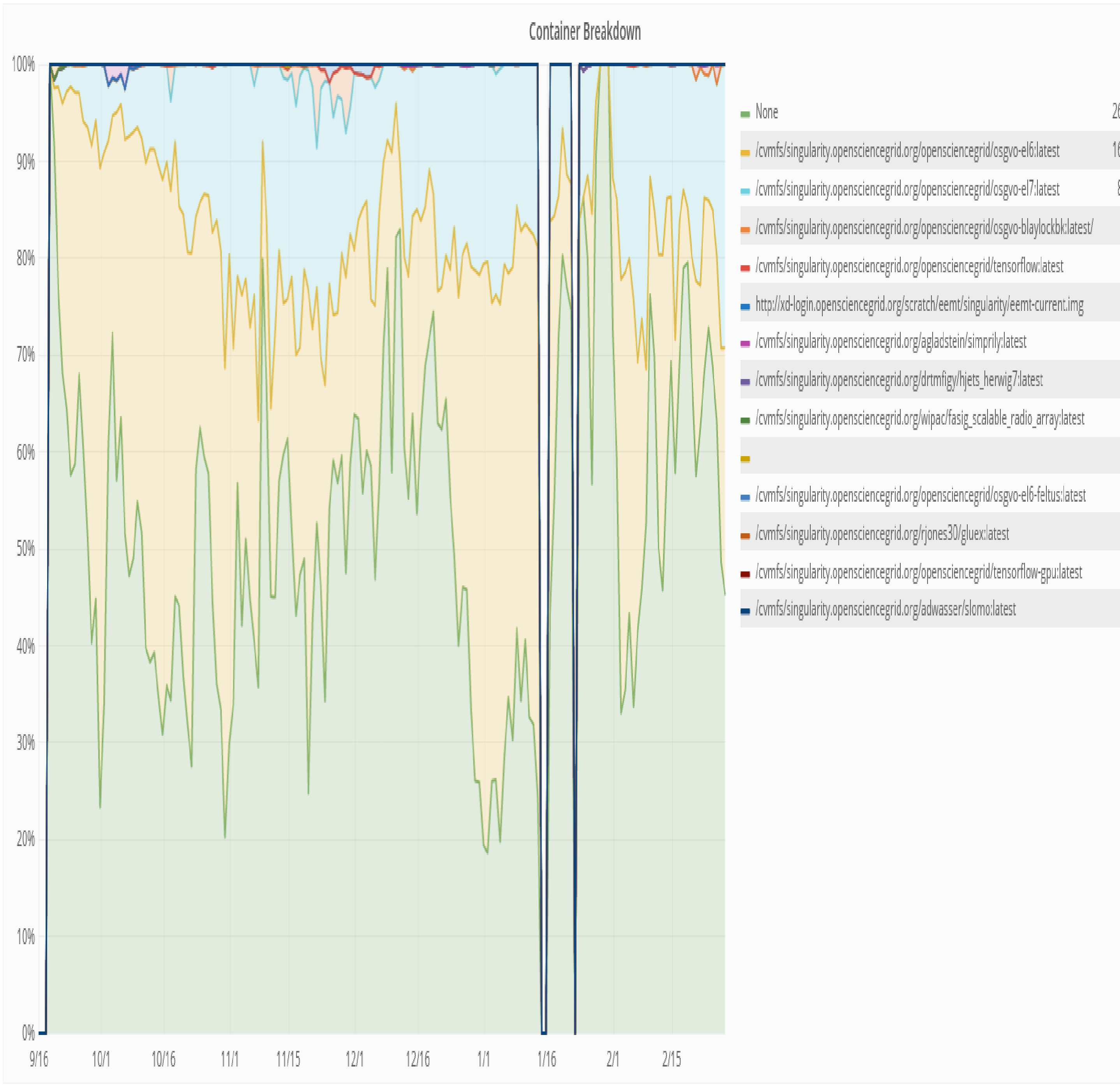
15.2.3 2.3 Container Statistics

These are for a subset of OSG, specifically for the OSG VO (Virtual Organization). Other VOs are also using containers.



One challenge when running these many container per day, across 100's of sites and 1000's of compute nodes, is how do we distribute and access containers without putting unnecessary load on Docker and Singularity hubs? More about this below.

The breakdown of jobs shows about half runs without containers, and the once running in containers are mostly doing so under the default images.

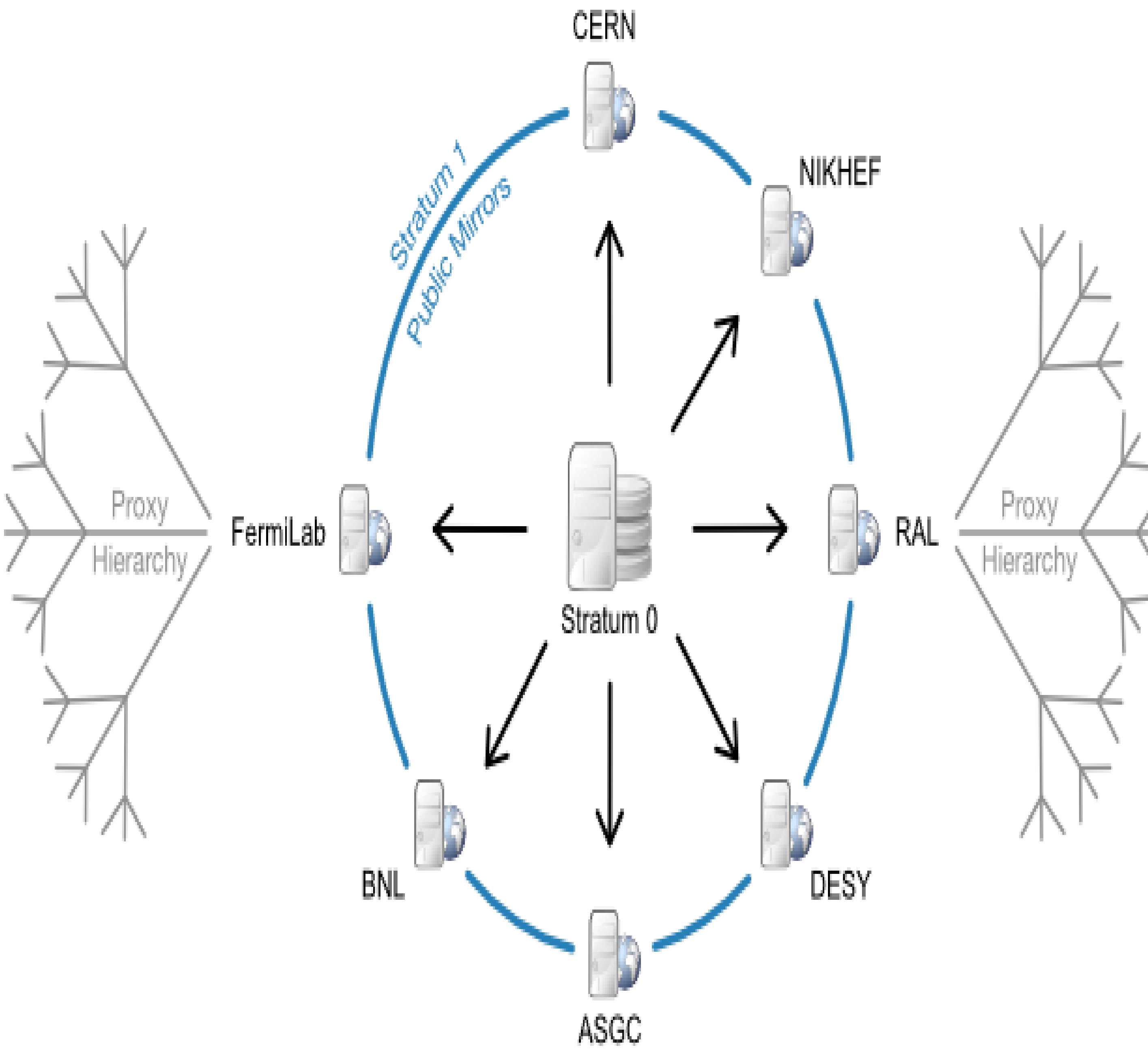


15.3 3. CVMFS

The CernVM File System (CVMFS) is a highly-scalable global filesystem optimized for global distribution of software. The CERN-based LHC experiments invested in this filesystem based on the experience of attempting to synchronize the install their complex application software stacks across hundreds of sites. Each release may contain tens of gigabytes of data across hundreds of thousands of files; a few dozen to a hundred releases might be active at any given time.

CVMFS is FUSE-based - a filesystem implemented in user space, not within the Linux kernel. It scales well because changes to each repository are only written to a single repository node and then distributed throughout the CVMFS content distribution network (a hierarchical set of web servers and HTTP caches). All writes are aggregated into a single transaction, making the rate of change relatively slow (typically, updates occur no faster than once every 15 minutes). Since file contents are immutable, CVMFS is able to use a content-addressed scheme and the corresponding HTTP objects immutable. Thus, the entire system is amenable to cache hierarchies.

CVMFS's original use case has significant parallels with distributing scientific containers: containers tend to be read-only, contain relatively large sets of software, and need to be accessed - without modification or corruption - at multiple sites.



OSG stores container images on CVMFS in extracted form. That is, we take the Docker image layers or the Singularity img/simg files and export them onto CVMFS. For example, *ls* on one of the containers looks similar to *ls /* on any Linux machine:

```
$ ls /cvmfs/singularity.opensciencegrid.org/opensciencegrid/osgvo-el7:latest/
cvmfs  host-libs  proc  sys  anaconda-post.log     lib64
dev    media      root  tmp  bin                   sbin
etc    mnt        run   usr  image-build-info.txt  singularity
home   opt        srv   var  lib
```

This is a very efficient way for use to distribute the images. Most jobs only need small parts of the actual image (as low as 25-100 MBs), and the CVMFS caching mechanism means those bits are aggressivly cached at both the site and node level.

15.3.1 3.1 cvmfs-singularity-sync

Information on how to register your image can be found on the Docker and Singularity Containers page in the OSG Help Desk. It says:

In order to be able to efficiently distribute the container images to a large of distributed compute hosts, OSG has chosen to host the images under CVMFS. Any image **publicly** available in Docker can be included for automatic syncing into the CVMFS repository. The result is an unpacked image under */cvmfs/singularity.opensciencegrid.org/*

To get your images included, please either create a git pull request against *docker_images.txt* in the cvmfs-singularity-sync repository, or contact user-support@opensciencegrid.org and we can help you.

Once your image has been registered, new versions pushed to Docker Hub will automatically be detected and CVMFS will be updated accordingly.

15.4 4. Exercise 1: Exploring Available Images

Log in via ssh to the training account provided on the slip of paper. *workflow.isi.edu* is a submit host for both Open Science Grid as well as a local HTCondor pool.

Look at at the directories and sub directories under */cvmfs/singularity.opensciencegrid.org*

```
$ ls /cvmfs/singularity.opensciencegrid.org/
```

Note how the directories in here relate to the *docker_images.txt* in the *cvmfs-singularity-sync* repository (link).

Let's explore an image which is different from the host operating system (CentOS 7). A good example of this is the TensorFlow image which is based on Ubuntu 16.04. Start an interactive shell and explore the environment, including verifying that TensorFlow is available and what version it is:

```
$ singularity shell /cvmfs/singularity.opensciencegrid.org/opensciencegrid/
↪tensorflow:latest/
Singularity: Invoking an interactive shell within container...

$ cat /etc/issue
Ubuntu 16.04.3 LTS

$ python3 -c 'import tensorflow as tf; print(tf.__version__)'
1.4.0

$ exit
```

Make sure you run *exit* as the remaining exercises will be run under the host operating system.

15.5 5. Exercise 2: Containerized Job - Default Image

Note: These exercises will continue to use the *workflow.isi.edu* submit host. If you want to use OSG for your research in the future, please sign up for an account on OSG Connect and then use the OSG Connect submit hosts.

More information on how to run jobs on OSG can be found in the OSG Connect Quick Start Guide

You will find an example HTCondor job under *~/ContainerCamp/OSG-02-Default-Image/*. Look at the content of *test-1.submit*

```
$ cd ~/ContainerCamp/OSG-02-Default-Image/
$ cat test-1.submit
```

The submit file specifies that we want Singularity, but not which image:

```
Requirements = HAS_SINGULARITY == True
```

Submit a job with:

```
$ condor_submit test-1.submit
```

Check on the job with *condor_q* or *condor_q -nobatch*:

```
$ condor_q
$ connor_q -nobatch
```

Once the job is complete, examine the created *job.*.output* file:

```
$ cat job.920697.0.output
Hello! I'm running on the site MWT2 on the node uct2-c235.mwt2.org
My Singularity image is /cvmfs/singularity.opensciencegrid.org/opensciencegrid/osgvo-
↪el6:latest
```

Open Science Grid has multiple default images! Currently, we have one for RHEL 6 and one for RHEL 7. The dynamic pool of resources contains a mix of these defaults. Most of the OSG users only cares about the base OS, and not whether it is a Singularity instance or not. A common requirments line is *Requirements = OSGVO_OS_STRING == "RHEL 6" && Arch == "X86_64" && HAS_MODULES == True* which maps to RHEL 6 (Singularity or native), on 64 bit host and which as a *modules* software. For fun, we can steer our jobs to a Singularity RHEL 7 instance. Let's take a look at *test-2.submit* requirments:

```
Requirements = HAS_SINGULARITY == True && OSGVO_OS_STRING == "RHEL 7"
```

Submit the job and examine the output just like for the first job:

```
$ condor_submit test-2.submit
```

15.6 6. Exercise 3: Containerized Job - Custom Image

To run a job with a custom image, the *+SingularityImage* is used in the submit file to specify what image should be used. Change your working directory into *~/ContainerCamp/OSG-03-Custom-Image* and look at the test-1.submit file

```
$ cd ~/ContainerCamp/OSG-03-Custom-Image/
$ cat test-1.submit
...
Requirements = HAS_SINGULARITY == True
+SingularityImage = "/cvmfs/singularity.opensciencegrid.org/opensciencegrid/
↪tensorflow:latest"
...
```

Using *+SingularityImage*, you can specify any image available under */cvmfs/singularity.opensciencegrid.org/*. Just like we did above, you can run this job and see what the output shows.

OSG also has **limited** support for pulling images directly from Singularity Hub, Docker Hub, or over http. This is pass-through functionallity of what Singularity can do (see the Singularity Quickstart Guide). This is a less efficient way to access the images, so only use this for small workloads. For larger workloads, please use the CVMFS approach. Also note that the APIs change, so pulling images directly from the hubs can be Singularity version sensitive. *test-2.submit* shows how we can specify that the job should use Singularity 2.4.2 and pull directly from the Singularity Hub. *+SingularityBindCVMFS = False* is required for images which do not have the /cvmfs directory and do not want to have CVMFS access.

```
$ cat test-2.submit
...
Requirements = HAS_SINGULARITY == True && OSG_SINGULARITY_VERSION == "2.4.2-dist"
+SingularityImage = "shub://singularityhub/ubuntu"
+SingularityBindCVMFS = False
...
```

CHAPTER 16

Pegasus Workflows with Application Containers

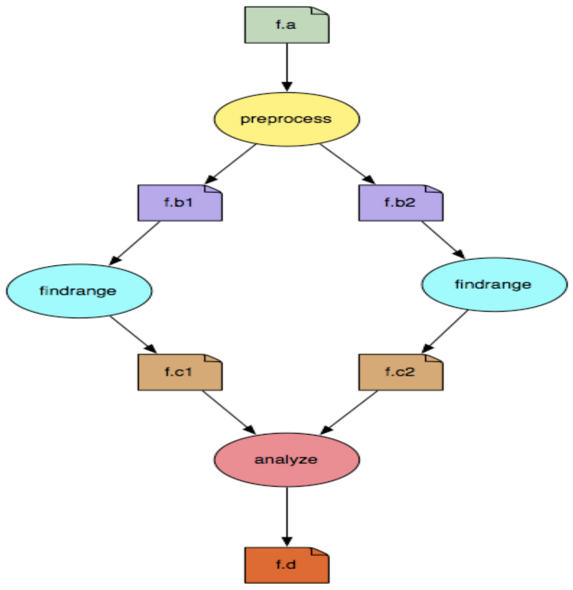
Note: This section contains an overview of scientific workflows, Pegasus, and how containers fits into Pegasus workflows. We will not have time to thoroughly cover all aspects of Pegasus - for future reference please see the user guide and self guided tutorial.

16.1 1. Prerequisites

ssh will be used to connect to a remote job submit host. Please ensure you have a ssh client installed. The instructors will supply a slip of paper with username, password and hostname during the session.

16.2 2. What are Scientific Workflows?

Scientific workflows allow users to easily express multi-step computational tasks, for example retrieve data from an instrument or a database, reformat the data, and run an analysis. A scientific workflow describes the dependencies between the tasks and in most cases the workflow is described as a directed acyclic graph (DAG), where the nodes are tasks and the edges denote the task dependencies. A defining property for a scientific workflow is that it manages data flow. The tasks in a scientific workflow can be everything from short serial tasks to very large parallel tasks (MPI for example) surrounded by a large number of small, serial tasks used for pre- and post-processing.



16.3 2. Pegasus Workflow Management System

Pegasus WMS is a configurable system for mapping and executing abstract application workflows over a wide range of execution environments including a laptop, a campus cluster, a Grid, or a commercial or academic cloud. Today, Pegasus runs workflows on Amazon EC2, Google Compute Engine, Open Science Grid, XSEDE, and campus clusters. One workflow can run on a single system or across a heterogeneous set of resources.

Pegasus WMS bridges the scientific domain and the execution environment by automatically mapping high-level workflow descriptions onto distributed resources. It automatically locates the necessary input data and computational resources necessary for workflow execution. Pegasus enables scientists to construct workflows in abstract terms without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). Pegasus WMS also bridges the current cyberinfrastructure by effectively coordinating multiple distributed resources. The input to Pegasus is a description of the abstract workflow in XML format.

Pegasus has a number of features that contribute to its useability and effectiveness.

- **Portability / Reuse**. User created workflows can easily be run in different environments without alteration. Pegasus currently runs workflows on top of Condor, Grid infrastrucutures such as Open Science Grid and TeraGrid, Amazon EC2, Nimbus, and many campus clusters. The same workflow can run on a single system or across a heterogeneous set of resources.
- **Performance**. The Pegasus mapper can reorder, group, and prioritize tasks in order to increase the overall workflow performance.
- **Scalability**. Pegasus can easily scale both the size of the workflow, and the resources that the workflow is distributed over. Pegasus runs workflows ranging from just a few computational tasks up to millions of tasks. The number of resources involved in executing a workflow can scale as needed without any impediments to performance.
- **Provenance**. By default, all jobs in Pegasus are launched via the kickstart process that captures runtime provenance of the job and helps in debugging. The provenance data is collected in a database, and the data can be summarised with tools such as pegasus-statistics, pegasus-plots, or directly with SQL queries.
- **Data Management**. Pegasus handles replica selection, data transfers and output registrations in data catalogs. These tasks are added to a workflow as auxilliary jobs by the Pegasus planner.

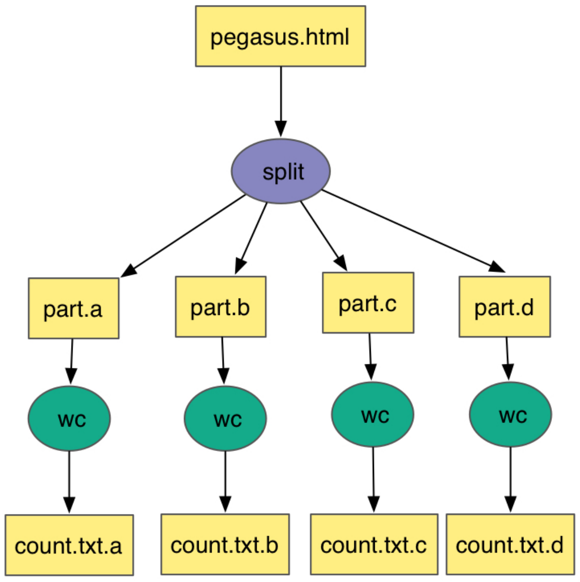
- **Reliability**. Jobs and data transfers are automatically retried in case of failures. Debugging tools such as pegasus-analyzer helps the user to debug the workflow in case of non-recoverable failures.
- **Error Recovery**. When errors occur, Pegasus tries to recover when possible by retrying tasks, by retrying the entire workflow, by providing workflow-level checkpointing, by re-mapping portions of the workflow, by trying alternative data sources for staging data, and, when all else fails, by providing a rescue workflow containing a description of only the work that remains to be done. It cleans up storage as the workflow is executed so that data-intensive workflows have enough space to execute on storage-constrained resource. Pegasus keeps track of what has been done (provenance) including the locations of data used and produced, and which software was used with which parameters.

16.4 3. Exercise 1: Without Containers

All of the example workflows described in the previous section can be generated with the pegasus-init command. For this tutorial we will be using the split workflow, which can be created like this:

```
$ pegasus-init split
Do you want to generate a tutorial workflow? (y/n) [n]: y
1: Local Machine
2: USC HPCC Cluster
3: OSG from ISI submit node
4: XSEDE, with Bosco
5: Bluewaters, with Glite
What environment is tutorial to be setup for? (1-5) [1]: 3
1: Process
2: Pipeline
3: Split
4: Merge
5: EPA (requires R)
What tutorial workflow do you want? (1-5) [1]: 3
Do you want to use Condor file transfers (y/n) [y]: y
Pegasus Tutorial setup for example workflow - split for execution on osg
```

The split workflow looks like this:



The input workflow description for Pegasus is called the DAX. It can be generated by running the *generate_dax.sh* script from the split directory, like this:

```
$ ./generate_dax.sh split.dax
Generated dax split.dax
```

This script will run a small Python program (*daxgen.py*) that generates a file with a .dax extension using the Pegasus Python API. Pegasus reads the DAX and generates an executable HTCondor workflow that is run on an execution site.

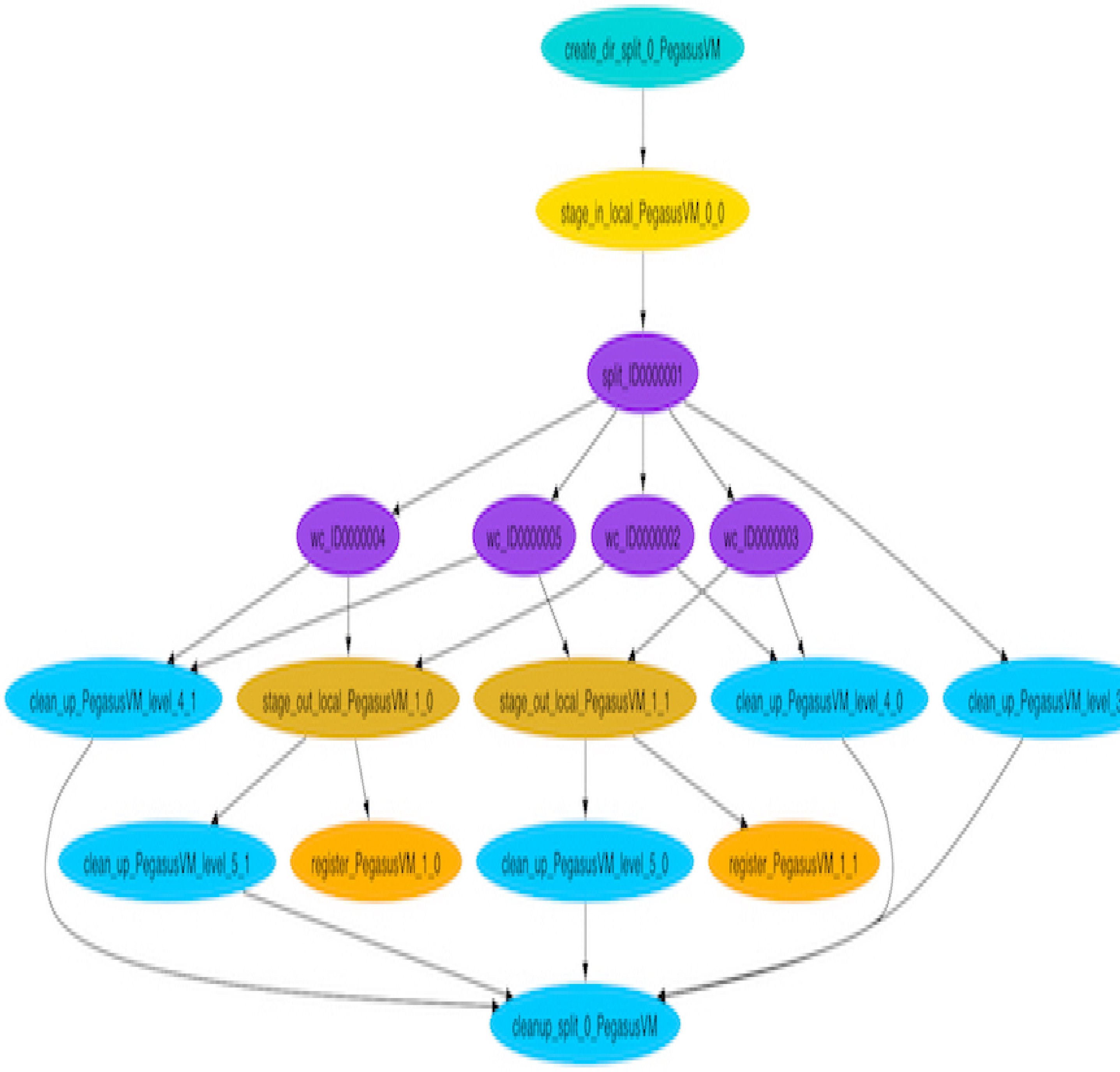
The *pegasus-plan* command is used to submit the workflow through Pegasus. The *pegasus-plan* command reads the input workflow (DAX file specified by –dax option), maps the abstract DAX to one or more execution sites, and submits the generated executable workflow to HTCondor. Among other things, the options to *pegasus-plan* tell Pegasus

- the workflow to run
- where (what site) to run the workflow
- the input directory where the inputs are placed
- the output directory where the outputs are placed

By default, the workflow is setup to run on the compute sites (i.e sites with handle other than "local") defined in the *sites.xml* file. In our example, the workflow will run on a site named "condorpool" in the *sites.xml* file.

```
$ ./plan_dax.sh split.dax

-----------------------------------------------------------------------
File for submitting this DAG to HTCondor       : split-0.dag.condor.sub
Log of DAGMan debugging messages               : split-0.dag.dagman.out
Log of HTCondor library output                 : split-0.dag.lib.out
Log of HTCondor library error messages         : split-0.dag.lib.err
Log of the life of condor_dagman itself        : split-0.dag.dagman.log
-----------------------------------------------------------------------
Submitting to condor split-0.dag.condor.sub
Submitting job(s).
1 job(s) submitted to cluster 920589.

Your workflow has been started and is running in the base directory:

  /split/submit/pegtrain50/pegasus/split/run0001

*** To monitor the workflow you can run ***

  pegasus-status -l /split/submit/pegtrain50/pegasus/split/run0001

*** To remove your workflow run ***

  pegasus-remove /split/submit/pegtrain50/pegasus/split/run0001
```

This is what the split workflow looks like after Pegasus has finished planning the DAX:


create_dir_split_0_PegasusVM
stage_in_local_PegasusVM_0_0
split_ID0000001
wc_ID0000004
wc_ID0000005
wc_ID0000002
wc_ID0000003
clean_up_PegasusVM_level_4_1
stage_out_local_PegasusVM_1_0
stage_out_local_PegasusVM_1_1
clean_up_PegasusVM_level_4_0
clean_up_PegasusVM_level_3
clean_up_PegasusVM_level_5_1
register_PegasusVM_1_0
clean_up_PegasusVM_level_5_0
register_PegasusVM_1_1
cleanup_split_0_PegasusVM

You can monitor the workflow with the *pegasus-status* command provided in the output of the *plan_dax.sh* command:

```
pegasus-status -l /split/submit/pegtrain50/pegasus/split/run0001
```

More details on how to run basic workflow can be found in the Pegasus Tutorial

16.5 4. Exercise 2: With Containers

Now when we have a basic understanding of what a Pegasus workflow looks like, let's use containers to run some real science codes. This example is based on IPAC's Montage toolkit, which is used to process and create astronomical image mosaics of from telescope images datasets. The workflow has a few software dependencies: Montage obviously, but also Python modules like AstroPy. These could be installed on the cluster you want to run the workflow on, but using containers makes it even easier!

Not only will we make the compute jobs run inside containers, but also the data find step needed to construct the workflow. IPAC provides services to list the images available for a given location in the sky (for example, see the documentation for mArchiveList). For this querying we will use the same container as the jobs will be using. To get started, clone the Montage workflow from GitHub, and run the data find step:

```
$ cd ~
$ git clone https://github.com/pegasus-isi/montage-workflow-v2.git
$ cd montage-workflow-v2
$ singularity exec \
              --bind $PWD:/srv --pwd /srv \
              shub://pegasus-isi/montage-workflow-v2 \
              /srv/montage-workflow.py \
                  --tc-target container \
                  --center "275.196290 -16.171530" \
                  --degrees 0.2 \
                  --band 2mass:j:green \
                  --band 2mass:h:blue \
                  --band 2mass:k:red
```

The three different *band* arguments specify different bands that we want to find images for, and map to *blue*, *green*, and *red* in to the final image. The output of the command should show a few images found for each band:

```
Progress |===================================| 100.0%

Adding band 1 (2mass j -> green)
Running sub command: mArchiveList 2mass j "275.196290 -16.171530" 0.284 0.284 data/1-
↪images.tbl
[struct stat="OK", module="mArchiveList", count=8]
Running sub command: cd data && mDAGTbls 1-images.tbl region-oversized.hdr 1-raw.tbl␣
↪1-projected.tbl 1-corrected.tbl
[struct stat="OK", count="8", total="8"]
Running sub command: cd data && mOverlaps 1-raw.tbl 1-diffs.tbl
[struct stat="OK", module="mOverlaps", count=13]

Adding band 2 (2mass h -> blue)
Running sub command: mArchiveList 2mass h "275.196290 -16.171530" 0.284 0.284 data/2-
↪images.tbl
[struct stat="OK", module="mArchiveList", count=8]
Running sub command: cd data && mDAGTbls 2-images.tbl region-oversized.hdr 2-raw.tbl␣
↪2-projected.tbl 2-corrected.tbl
[struct stat="OK", count="8", total="8"]
Running sub command: cd data && mOverlaps 2-raw.tbl 2-diffs.tbl
[struct stat="OK", module="mOverlaps", count=13]

Adding band 3 (2mass k -> red)
Running sub command: mArchiveList 2mass k "275.196290 -16.171530" 0.284 0.284 data/3-
↪images.tbl
[struct stat="OK", module="mArchiveList", count=8]
Running sub command: cd data && mDAGTbls 3-images.tbl region-oversized.hdr 3-raw.tbl␣
↪3-projected.tbl 3-corrected.tbl
```

(continues on next page)

(continued from previous page)

```
[struct stat="OK", count="8", total="8"]
Running sub command: cd data && mOverlaps 3-raw.tbl 3-diffs.tbl
[struct stat="OK", module="mOverlaps", count=13]
```

The *data/* directory contains the imformation about the input images, the generated workflow (*data/montage.dax*) and the transformation catalog (*data/tc.txt*) which tells Pegasus where software is available. A job in the *data/montage.dax* file might look like:

```
<job id="ID0000001" name="mProject">
    <argument>-X <file name="2mass-atlas-990502s-j1420198.fits"/> <file name="p2mass-
↪atlas-990502s-j1420198.fits"/> <file name="region-oversized.hdr"/></argument>
    <uses name="region-oversized.hdr" link="input"/>
    <uses name="2mass-atlas-990502s-j1420198.fits" link="input"/>
    <uses name="p2mass-atlas-990502s-j1420198.fits" link="output" transfer="false"/>
    <uses name="p2mass-atlas-990502s-j1420198_area.fits" link="output" transfer="false
↪"/>
</job>
```

data/tc.txt has the specification on how *mProject* can be executed:

```
tr mProject {
  site condor_pool {
    type "INSTALLED"
    container "montage"
    pfn "file:///opt/Montage/bin/mProject"
    profile pegasus "clusters.size" "3"
  }
}
```

Note the *container "montage"* part - this is a reference to the top of the file which has:

```
cont montage {
   type "singularity"
   image "shub://pegasus-isi/montage-workflow-v2"
   profile env "MONTAGE_HOME" "/opt/Montage"
}
```

Which is the same container we used for the data find step. Note that container images is just like any other piece of data to Pegasus. In this case, the image will be downloaded **once** from the Singularity Hub, and then shipped around to the jobs with the same mechanism as any other data in the workflow.

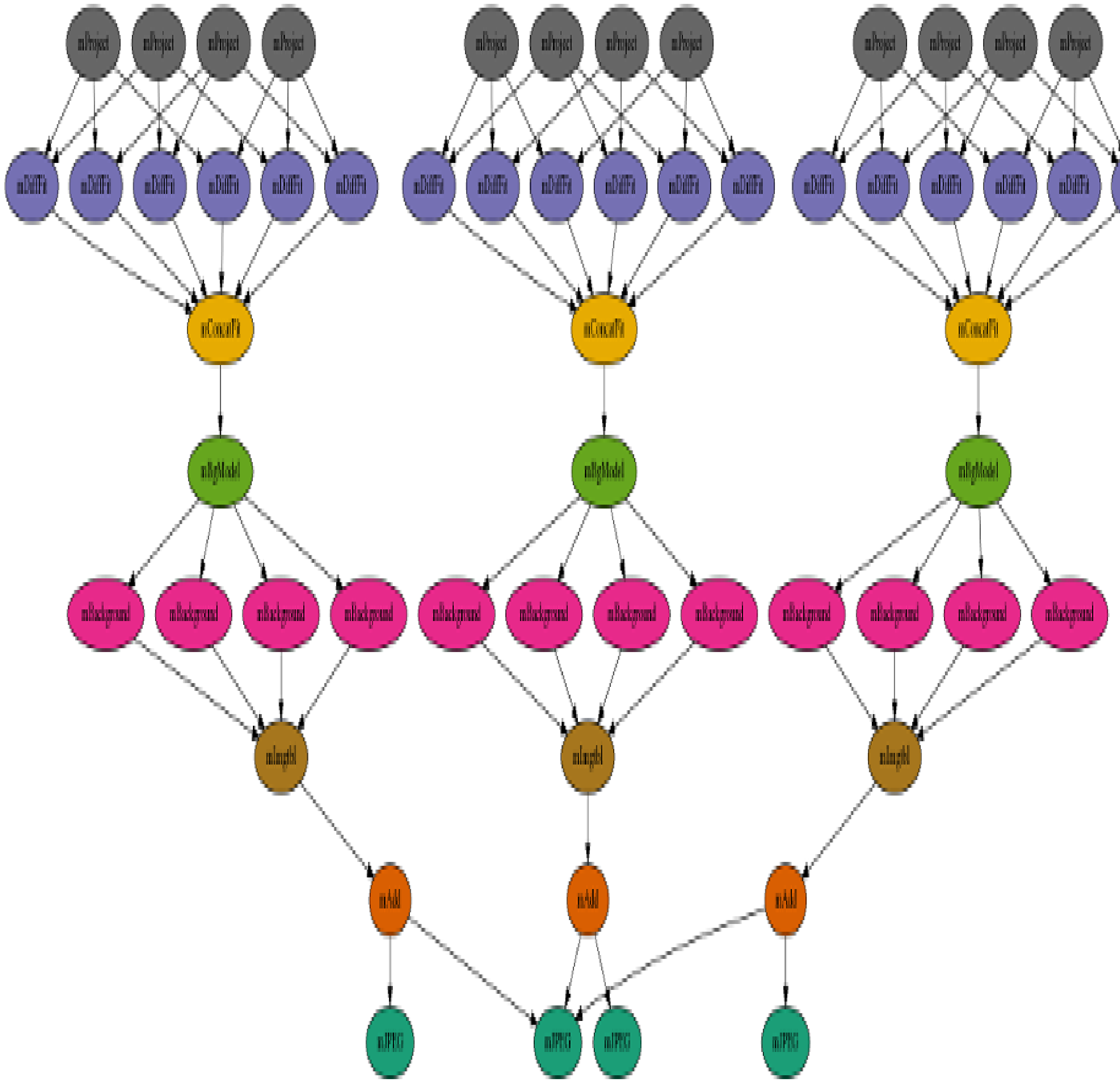
There is currently a small issue by running the data find step inside a container - the paths for the files are based on paths in the container which are different from what Pegasus expects on the submit host. The following command adjusts those paths:

```
$ perl -p -i -e "s;/srv/data;$PWD/data;g" data/rc.txt
```

Now we are are ready to plan and submit the workflow:

```
$ pegasus-plan \
        --dir work \
        --relative-dir `date +'%s'` \
        --dax data/montage.dax \
        --sites condor_pool \
        --output-site local \
        --submit
```

The workflow will looks something like this:



The first level reprojects the input images to a common projection. The images are then fitted together. A background correction is applied so that the the final image will be seamless. The last step is to take the 3 different color bands, and add them together into a final output image:

To see how Pegasus handled the container in this case, let's look at some plumming for one of the *mProject* job. The HTCondor submit file can be seen with:

```
$ cat `find . -name mProject_ID0000002.sub`
```

Look at the *transfer_input_files* attribute line, and specifically for the *montage.simg* file. It is transferred together with all the other inputs for the job:

```
transfer_input_files = region-oversized.hdr,2mass-atlas-990502s-j1350092.fits,montage.
↪simg,/opt/training/pegasus-4.8.2dev/share/pegasus/sh/pegasus-lite-common.sh,/
↪scitech/home/pegtrain99/montage-workflow-v2/work/1520295762/pegasus-worker-4.8.2dev-
↪x86_64_rhel_7.tar.gz
```

(continues on next page)

(continued from previous page)

Looking at the corresponding *.sh* file we can see how Pegasus executed the container:

```
$ cat `find . -name mProject_ID0000002.sh`
...
singularity exec --pwd /srv --scratch /var/tmp --scratch /tmp --home $PWD:/srv␣
↪montage.simg ./mProject_ID0000002-cont.sh
...
```

The *./mProject_ID0000002-cont.sh* is a script generated at runtime, containing the execution of the user codes.

CHAPTER 17

Distributed Computing with Makeflow and Work Queue

17.1 1. Prerequisites

'ssh' will be used to connect to a remote job submit host. Please ensure you have a ssh client installed. The instructors will supply a slip of paper with username, password and hostname during the session.

This tutorial will also be linked to from our tutorial webpage: http://ccl.cse.nd.edu/software/tutorials/cyversecc18/

Our website is located at: http://ccl.cse.nd.edu/

You can get the slides from this talk there as well as additional material for our tools.

17.2 2. Cooperative Computing Lab

The CCL designs software that enables our collaborators to easily harness large scale distributed systems such as clusters, clouds, and grids. We perform fundamental computer science research that enables new discoveries through computing in fields such as physics, chemistry, bioinformatics, biometrics, and data mining.

The software suite we write and maintain is the CCTools software package.

- **Makeflow**. A portable workflow manager. Link
- **Work Queue**. A lightweight distributed execution system. Link
- **Parrot**. A personal user-level virtual file system. Link
- **Chirp**. A user-level distributed filesystem. Link
- **Specialized Software**. A selection of applications tailored to specific compuation tasks.

17.3 2. Makeflow

Makeflow is a workflow system for executing large complex workflows on clusters, clouds, and grids.

- **Makeflow is easy to use.** The Makeflow language is similar to traditional Make, so if you can write a Makefile, then you can write a Makeflow. A workflow can be just a few commands chained together, or it can be a complex application consisting of thousands of tasks. It can have an arbitrary DAG structure and is not limited to specific patterns.
- **Makeflow is production-ready.** Makeflow is used on a daily basis to execute complex scientific applications in fields such as data mining, high energy physics, image processing, and bioinformatics. It has run on campus clusters, the Open Science Grid, NSF XSEDE machines, NCSA Blue Waters, and Amazon Web Services. Here are some real examples of workflows used in production systems:
- **Makeflow is portable.** A workflow is written in a technology neutral way, and then can be deployed to a variety of different systems without modification, including local execution on a single multicore machine, public cloud services such as Amazon EC2 and Amazon Lambda, batch systems like HTCondor, SGE, PBS, Torque, SLURM, or the bundled Work Queue system. Makeflow can also easily run your jobs in a container environment like Docker or Singularity on top of an existing batch system. The same specification works for all systems, so you can easily move your application from one system to another without rewriting everything.
- **Makeflow is powerful.** Makeflow can handle workloads of millions of jobs running on thousands of machines for months at a time. Makeflow is highly fault tolerant: it can crash or be killed, and upon resuming, will reconnect to running jobs and continue where it left off. A variety of analysis tools are available to understand the performance of your jobs, measure the progress of a workflow, and visualize what is going on.

17.4 2. Work Queue

Work Queue is a framework for building large master-worker applications that span thousands of machines drawn from clusters, clouds, and grids. Work Queue applications are written in C, Perl, or Python using a simple API that allows users to define tasks, submit them to the queue, and wait for completion. Tasks are executed by a standard worker process that can run on any available machine. Each worker calls home to the master process, arranges for data transfer, and executes the tasks. The system handles a wide variety of failures, allowing for dynamically scalable and robust applications.

Work Queue has been used to write applications that scale from a handful of workstations up to tens of thousands of cores running on supercomputers. Examples include Lobster, NanoReactors, ForceBalance, Accelerated Weighted Ensemble, the SAND genome assembler, the Makeflow workflow engine, and the All-Pairs and Wavefront abstractions. The framework is easy to use, and has been used to teach courses in parallel computing, cloud computing, distributed computing, and cyberinfrastructure at the University of Notre Dame, the University of Arizona, and the University of Wisconsin - Eau Claire.

17.5 3. Makeflow Tutorial

This tutorial goes through the installation process of CCTools, the creation and running of a Makeflow, and how to use Makeflow in conjunction with Work Queue to leverage different execution resources for your execution. More information can be found a http://ccl.cse.nd.edu/. For specific information on Makeflow execution see http://ccl.cse.nd.edu/software/manuals/makeflow.html and Work Queue see http://ccl.cse.nd.edu/software/manuals/workqueue.html.

17.6 3.1. Running on Atmosphere/Jetstream

To start out we are going to launch an instance:

We are going to be using an Ubuntu instance with Docker already installed: Ubuntu 16.04 Devel and Docker v.1.13

Please note you should use images of at least **Medium** size.

Once the instance is up, we are going to add a few packages to allow for easy installation. Most of these packages are already installed on batch submission sites, but possibly not in all Jetstream instances.

```
$ sudo apt-get install zlib1g-dev libncurses5-dev g++
```

Additionally, we are going to add our current user to the docker group:

```
$ sudo usermod -aG docker ${USER}
```

We are also going to install Singularity if you have not done so yet. This should be done using the provided ansible script:

```
$ ezs
```

After adding this log out and back in.

```
$ exit
```

Now re-open the in web-shell. Once you are logged back in, we are going to pull the docker image we will use today:

```
$ docker pull nekelluna/ccl_makeflow_examples
$ docker save -o mfe.tar nekelluna/ccl_makeflow_examples
$ singularity pull docker://nekelluna/ccl_makeflow_examples
```

Note: If you would like to test this out with Work Queue on another machine, now is a great time to launch and do these setup steps on each machine. `Hint hint` you should do this.

17.7 3.2. Download and Installation

Once you have a shell, download and install the CCTools software in your home directory in one of two ways:

To build our latest release:

```
$ wget http://ccl.cse.nd.edu/software/files/cctools-6.2.6-source.tar.gz
$ tar zxpvf cctools-6.2.6-source.tar.gz
$ cd cctools-6.2.6-source
$ ./configure --prefix $HOME/cctools --tcp-low-port 9000 --tcp-high-port 9500
$ make
$ make install
$ cd $HOME
```

If you use bash then do this to set your path:

```
$ export PATH=$HOME/cctools/bin:$PATH
```

If you use tcsh instead, then do this:

```
$ setenv PATH $HOME/cctools/bin:$PATH
```

Now double check that you can run the various commands, like this:

```
$ makeflow -v
$ work_queue_worker -v
$ work_queue_status
```

17.8 3.3. Getting Makeflow-Examples

As a good reference point for workflow design and examples we are going to use our Makeflow Examples repository.

```
$ git clone https://github.com/cooperative-computing-lab/makeflow-examples.git
  -- or --
$ wget https://github.com/cooperative-computing-lab/makeflow-examples/archive/master.
↪zip
```

If you used wget to pull down the zip file remember to unzip and enter this directory:

```
$ unzip master.zip
$ mv master makeflow-examples
$ cd makeflow-examples
```

17.9 4.1. Makeflow Example

Let's begin by using Makeflow to run a handful of simulation codes. First, make and enter a clean directory to work in inside of `makeflow-examples`:

```
$ cd $HOME/makeflow-examples
$ mkdir tutorial
$ cd tutorial
```

Download this program, which performs a highly sophisticated simulation of black holes colliding together:

```
$ wget http://ccl.cse.nd.edu/software/tutorials/cyversecc18/simulation.py
```

Try running it once, just to see what it does:

```
$ chmod 755 simulation.py
$ ./simulation.py 5
```

Now, let's use Makeflow to run several simulations. Create a file called `example.makeflow` and paste the following text into it:

```
input.txt:
    LOCAL /bin/echo "Hello Makeflow!" > input.txt

output.1: simulation.py input.txt
    ./simulation.py 1 < input.txt > output.1

output.2: simulation.py input.txt
    ./simulation.py 2 < input.txt > output.2

output.3: simulation.py input.txt
    ./simulation.py 3 < input.txt > output.3

output.4: simulation.py input.txt
    ./simulation.py 4 < input.txt > output.4
```

To run it on your local machine, one job at a time:

```
$ makeflow example.makeflow -j 1
```

Note that if you run it a second time, nothing will happen, because all of the files are built:

```
$ makeflow example.makeflow
$ makeflow: nothing left to do
```

Use the -c option to clean everything up before trying it again:

```
$ makeflow -c example.makeflow
```

Here are some other options for built-in batch systems:

```
$ makeflow -T slurm example.makeflow
$ makeflow -T torque example.makeflow
$ makeflow -T sge example.makeflow
```

17.10 4.2. Running Makeflow with Work Queue

You will notice that a workflow can run very slowly if you submit each job individually. To get around this limitation, we provide the Work Queue system. This allows Makeflow to function as a master process that quickly dispatches work to remote worker processes.

```
$ makeflow -c example.makeflow
$ makeflow -T wq example.makeflow -p 0
listening for workers on port XXXX.
...
```

Now open up another shell and run a single worker process:

```
$ work_queue_worker crcfe01.crc.nd.edu XXXX
```

Go back to your first shell and observe that the makeflow has finished. Of course, remembering port numbers all the time gets old fast, so try the same thing again, but using a project name:

```
$ makeflow -c example.makeflow
$ makeflow -T wq example.makeflow -N project-$USER
listening for workers on port XXXX
...
```

Now open up another shell and run your worker with a project name:

```
$ work_queue_worker -N project-$USER
```

17.11 5. Using Containers with Makeflow

We are going to start using Containers in the Makeflow by showing the different configurations that we talked about in the slides. There is a simple, 1 rule, makeflow that we will use to show these:

```
hello.out:
    echo "hello, world!" > hello.out
```

The first configuration we discussed would be to run both the Makeflow and the Worker inside of container to allow for a consistent environment.

We will not do this here, as that is extremely similar to running in Atmosphere/Jetstream to begin with. This is great way to test out different software configurations when determining what is needed for a workflow and how different software will interact.

The second configuration is to run each task inside of separate containers. This configuration is useful for specializing the configuration each task uses and not assuming the execution site has any software requirements aside from docker or singularity.

Assuming we are wrapping each task in a container, there are two ways to do this in Makeflow. The first is to manually add the container to your command. This allows for precise control of how the task is executed and in which container this occurs. We will show this now:

We are going to look at what the hello-containers folder:

```
$ cd $HOME/makeflow-examples
$ cd hello-containers
```

Inside of the `hello-containers` folder, there is a python script, `hello_world_creator.py`, that will create a simple hello world example which uses a container:

To test with Docker:

```
$ python hello_world_creator.py --docker nekelluna/ccl_makeflow_examples
```

To test with Singularity

```
$ ln -s $HOME/ccl_makeflow_examples.simg ccl_makeflow_examples.simg
$ python hello_world_creator.py --singularity ccl_makeflow_examples.simg
```

After running these, look at `hello_world.mf` and see how the above run has been wrapped by the container command. Now we are just going to run this locally:

```
$ makeflow hello_world.mf -T local
```

Now, instead of wrapping each task by hand, we are going to assume that each task will use the same container. For this we will use Makeflow's built in support for containers. We will assume that the above steps for either docker or singularity have been done:

```
$ cd $HOME/makeflow-examples
$ cd hello-world
```

We are going to start from the existing `hello-world` example. To run Makeflow with either docker or singularity we specify the container in the arguments:

Docker:

```
$ ln -s $HOME/mfe.tar mfe.tar
$ makeflow hello_world.mf --docker=nekelluna/ccl_makeflow_examples --docker-tar=mfe.
↪tar
```

Singularity:

```
$ ln -s $HOME/ccl_makeflow_examples.simg ccl_makeflow_examples.simg
$ makeflow hello_world.mf --singularity=ccl_makeflow_examples.simg
```

We have three additional examples that will work with the above provided container.

- *5.1. BLAST in a Container*
- *5.2. BWA in a Container*

- *5.3. Text Analysis in a Container*

Each of these examples may have a small amount of setup to pull/compile the software needed.

17.12 5.1. BLAST in a Container

BLAST is a common bioinformatic application used for determining alignment of a query dataset with a known reference set. BLAST compares each line independently of each other, allowing for clear parallelism opportunities.

```
$ cd $HOME/makeflow-examples
$ cd blast
```

We use an older BLAST executable for this example, as this creation script has not been changed. These commands pull down the executable and a reference database.

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/blast/executables/legacy.NOTSUPPORTED/2.2.26/blast-
↪2.2.26-x64-linux.tar.gz
$ tar xvzf blast-2.2.26-x64-linux.tar.gz
$ cp blast-2.2.26/bin/blastall .
$ wget ftp://ftp.ncbi.nlm.nih.gov/blast/db/nt.44.tar.gz
$ mkdir nt
$ tar -C nt -xvzf nt.44.tar.gz
```

We are now going to generate a random data set to align with the reference:

```
$ ./fasta_generator 200 1000 > test.fasta
```

Based on the generated data, we will now write a makeflow:

```
$ ./makeflow_blast -d nt -i test.fasta -p blastn --num_seq 5 --makeflow blast_test.mf
```

Assuming you have already pulled the images needed for either *singularity* or *docker* we will run them similarly to how it was done above:

Docker:

```
$ ln -s $HOME/mfe.tar mfe.tar
$ makeflow blast_test.mf --docker=nekelluna/ccl_makeflow_examples --docker-tar=mfe.tar
```

Singularity:

```
$ ln -s $HOME/ccl_makeflow_examples.simg ccl_makeflow_examples.simg
$ makeflow blast_test.mf --singularity=ccl_makeflow_examples.simg
```

17.13 5.2. BWA in a Container

BWA is similar to BLAST in that it is a bioinformatics tool that aligns a query dataset with a reference dataset. BWA does not operate on highly structured reference data like BLAST, but uses a fasta or fastq data file for both the query and reference.

```
$ cd $HOME/makeflow-examples
$ cd bwa
```

We will download and compile the software:

```
$ git clone https://github.com/lh3/bwa bwa-src
$ cd bwa-src
$ make
$ cp bwa ..
$ cd ..
```

Create the data we will use for the analysis:

```
$ ./fastq_generate.pl 10000 1000 > ref.fastq
$ ./fastq_generate.pl 1000 100 ref.fastq > query.fastq
```

The first line creates the reference dataset and the second will create a query dataset based on a portion of the provided reference dataset. This allows us to guarantee there will be some overlap and data analysis at each step for this example.

Now we will create the makeflow based on the input dataset:

```
$ ./make_bwa_workflow --ref ref.fastq --query query.fastq --num_seq 100 > bwa.mf
```

Again assuming that the docker and singularity images have been pulled down, run the makeflow:

Docker:

```
$ ln -s $HOME/mfe.tar mfe.tar
$ makeflow bwa.mf --docker=nekelluna/ccl_makeflow_examples --docker-tar=mfe.tar
```

Singularity:

```
$ ln -s $HOME/ccl_makeflow_examples.simg ccl_makeflow_examples.simg
$ makeflow bwa.mf --singularity=ccl_makeflow_examples.simg
```

17.14 5.3. Text Analysis in a Container

The test analysis example that we are providing is a simple makelfow that analyzes a set of Shakespeare's plays. This workflow gives an example of using Makeflow to parallelize a text search through a collection of William Shakespeare's plays. Makeflow will download the plays, package up the version of Perl at the location Makeflow is running, and run a text analysis Perl script in parallel to figure out which character had the most dialogue out of the plays selected.

```
$ cd $HOME/makeflow-examples
$ cd shakespeare
```

This workflow relys on Perl and CCTools being installed, so there is no further setup needed.

Docker:

```
$ ln -s $HOME/mfe.tar mfe.tar
$ makeflow shakespeare.makeflow --docker=nekelluna/ccl_makeflow_examples --docker-
↪tar=mfe.tar
```

Singularity:

```
$ ln -s $HOME/ccl_makeflow_examples.simg ccl_makeflow_examples.simg
$ makeflow shakespeare.makeflow --singularity=ccl_makeflow_examples.simg
```

CHAPTER 18

Introduction to Biocontainers

BioContainers is a community-driven project that provides the infrastructure and basic guidelines to create, manage and distribute Bioinformatics containers with special focus in Proteomics, Genomics, Transcriptomics and Metabolomics. BioContainers is based on the popular frameworks of Docker.

BioContainers Goals:

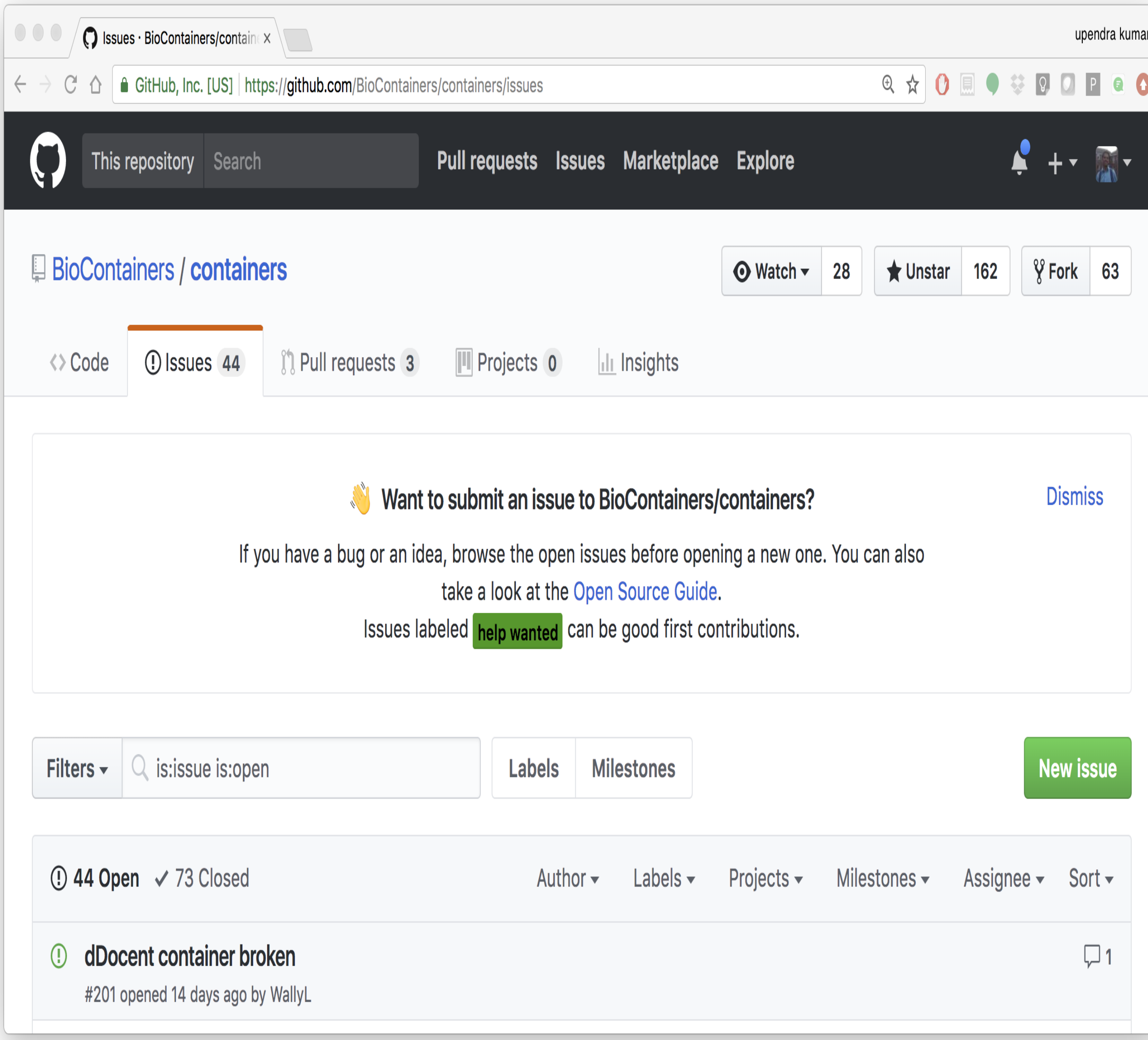
- Provide a base specification and images to easily build and deploy new bioinformatics/proteomics software including the source and examples.
- Provide a series of containers ready to be used by the bioinformatics community (https://github.com/BioContainers/containers).
- Define a set of guidelines and specifications to build a standardized container that can be used in combination with other containers and bioinformatics tools.
- Define a complete infrastructure to develop, deploy and test new bioinformatics containers using continuous integration suites such as Travis Continuous Integration (https://travisci. org/), Shippable (https://app.shippable.com/) or manually built solutions.
- Provide support and help to the bioinformatics community to deploy new containers for researchers that do not have bioinformatics support.
- Provide guidelines and help on how to create reproducible pipelines by defining, reusing and reporting specific container versions which will consistently produce the exact same result and always be available in the history of the container.
- Coordinate and integrate developers and bioinformaticians to produce best practice of documentation and software development.

18.1 Developing biocontainers

18.1.1 1. Docker based Biocontainers

1.1 - How to Request a Biocontainer?

Users can request a container by opening an issue in the containers repository



Want to submit an issue to BioContainers/containers?

If you have a bug or an idea, browse the open issues before opening a new one. You can also take a look at the Open Source Guide.

Issues labeled help wanted can be good first contributions.

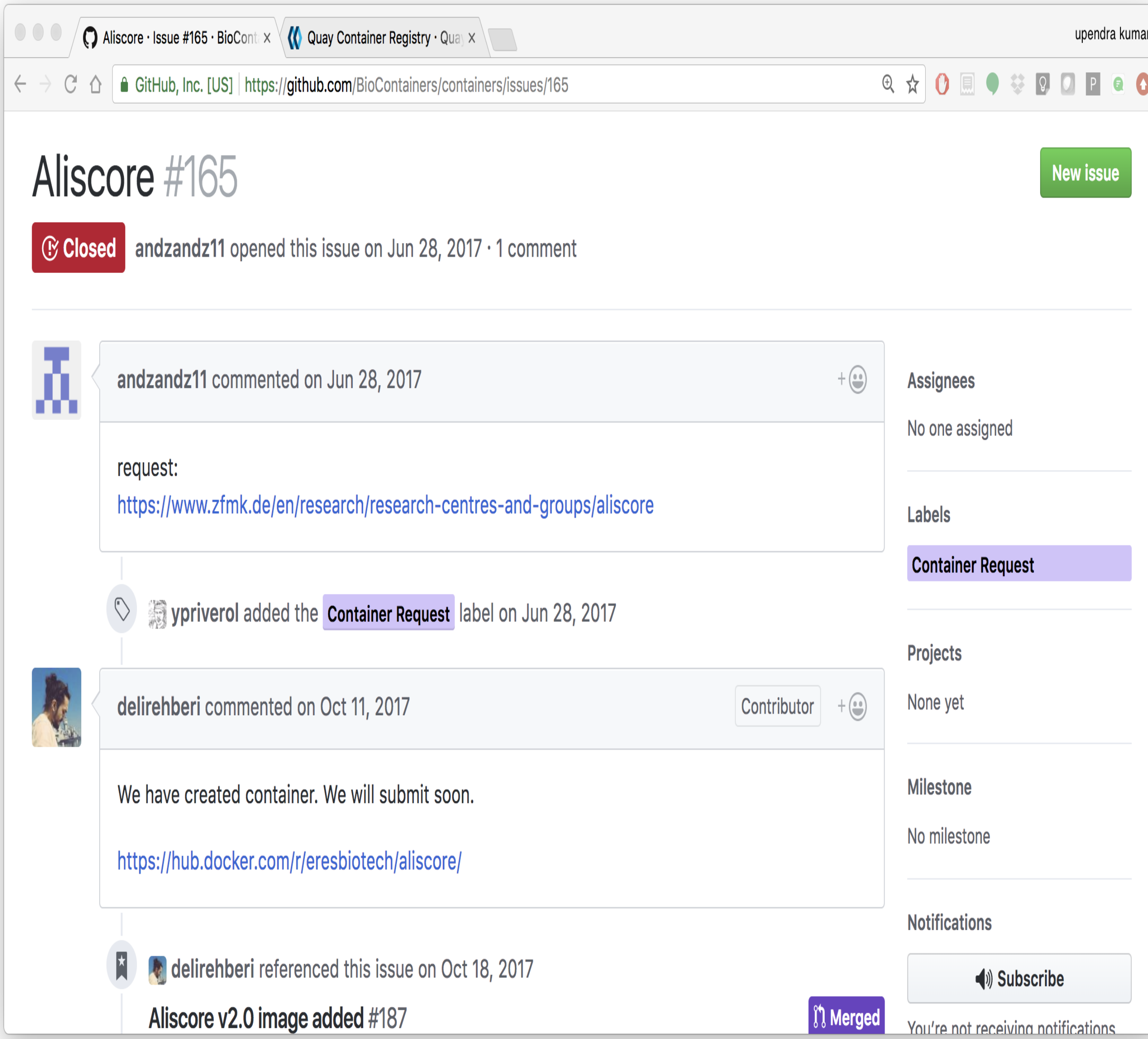
dDocent container broken

#201 opened 14 days ago by WallyL

The issue should contains the name of the software, the url of the code or binary to be package and information about the software. When the containers is deployed and fully functional, the issue will be closed by the developer or the contributor to BioContainers.

1.1.1 - Use a BioContainer

When a container is deployed and the developer closes the issue in GitHub the user received a notification that the container is ready.


Aliscore · Issue #165 · BioCont
Quay Container Registry · Qua
upendra kuma
GitHub, Inc. [US] https://github.com/BioContainers/containers/issues/165
New issue

Aliscore #165

Closed andzandz11 opened this issue on Jun 28, 2017 · 1 comment

andzandz11 commented on Jun 28, 2017

request:
https://www.zfmk.de/en/research/research-centres-and-groups/aliscore

ypriverol added the Container Request label on Jun 28, 2017

delirehberi commented on Oct 11, 2017 Contributor

We have created container. We will submit soon.

https://hub.docker.com/r/eresbiotech/aliscore/

delirehberi referenced this issue on Oct 18, 2017

Aliscore v2.0 image added #187 Merged

Assignees
No one assigned

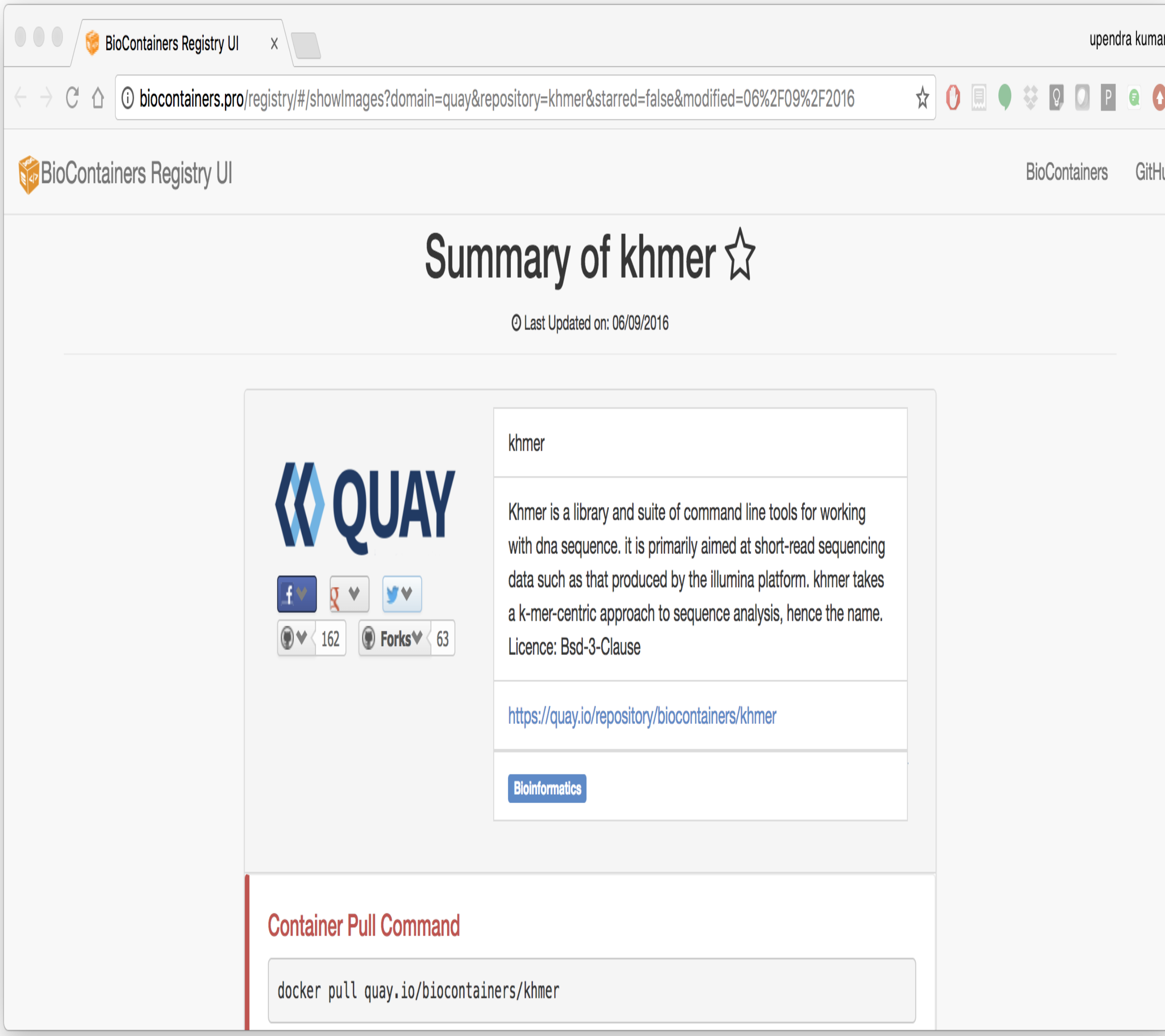
Labels
Container Request

Projects
None yet

Milestone
No milestone

Notifications
Subscribe

You're not receiving notifications


BioContainers Registry UI
upendra kuma
biocontainers.pro/registry/#/showImages?domain=quay&repository=khmer&starred=false&modified=06%2F09%2F2016
BioContainers Registry UI
BioContainers
GitH
QUAY
162
Forks
63

Summary of khmer

Last Updated on: 06/09/2016

khmer

Khmer is a library and suite of command line tools for working with dna sequence. it is primarily aimed at short-read sequencing data such as that produced by the illumina platform. khmer takes a k-mer-centric approach to sequence analysis, hence the name. Licence: Bsd-3-Clause

https://quay.io/repository/biocontainers/khmer

Bioinformatics

Container Pull Command

```
docker pull quay.io/biocontainers/khmer
```

The user can then use *docker pull* or *docker run* for the corresponding container from *quay.io/biocontainers*. For example

```
docker pull quay.io/biocontainers/khmer:2.1.2--py36_0
```

Note: Reporting a problem with a container: If the user find a problem with a container an issue should be open in the container repository, the user should use the broken tag (see tags). Developers of the project will pick-up the issue and deploy a new version of the container. A message will be delivered when the containers has been fixed.

1.2 - Create a Dockerfile for Biocontainer

If you are familiar with Docker (which you are by now!), then instead of requesting a biocontainer, you can create a Dockerfile and then submit the Dockerfile for biocontainer

- BioContainers dockerfile template

Note: Please always follow the best practices and help pages using input and output files information.

Below is the complete example of a BioContainers Dockerfile:

```
# Base Image
FROM biocontainers/biocontainers:latest

# Metadata
LABEL base.image="biocontainers:latest"
LABEL version="3"
LABEL software="Comet"
LABEL software.version="2016012"
LABEL description="an open source tandem mass spectrometry sequence database search␣
↪tool"
LABEL website="http://comet-ms.sourceforge.net/"
LABEL documentation="http://comet-ms.sourceforge.net/parameters/parameters_2016010/"
LABEL license="http://comet-ms.sourceforge.net/"
LABEL tags="Proteomics"

# Maintainer
MAINTAINER Felipe da Veiga Leprevost <felipe@leprevost.com.br>

USER biodocker

RUN ZIP=comet_binaries_2016012.zip && \
  wget https://github.com/BioDocker/software-archive/releases/download/Comet/$ZIP -O /
↪tmp/$ZIP && \
  unzip /tmp/$ZIP -d /home/biodocker/bin/Comet/ && \
  chmod -R 755 /home/biodocker/bin/Comet/* && \
  rm /tmp/$ZIP

RUN mv /home/biodocker/bin/Comet/comet_binaries_2016012/comet.2016012.linux.exe /home/
↪biodocker/bin/Comet/comet

ENV PATH /home/biodocker/bin/Comet:$PATH

WORKDIR /data/

CMD ["comet"]
```

1.2.1 - Run it!, Test it! Once the container is ready you should test it, try to run your program using the run command, check if all its functionalities are in order.

1.2.2 - Contribute if everything looks OK. You can contribute to the BioContainers project by sending your Dockerfile.

18.1.2 2. Bioconda based Biocontainers

In contrast to traditional Biocontainers, Bioconda based Biocontainers offers a very easy way to create efficient containers that are minimal in size, tested and not rely on writing a Dockerfile.

The preferred way to do this is to write a conda package and submit this it the BioConda communtiy. As soon as your PR is merged and continues integration testing was successful, a container is built and publish it at quay.io.

In summary, a BioConda recipe should contain the following parts:

- Source URL is stable (details)
- md5 or sha256 hash included for source download (details)
- Appropriate build number (details)
- .bat file for Windows removed (details)
- Remove unnecessary comments (details)
- Adequate tests included
- Files created by the recipe follow the FSH (details)
- License allows redistribution and license is indicated in meta.yaml
- Package does not already exist in the defaults, r, or conda-forge channels with some exceptions (details)
- Package is appropriate for bioconda
- If the recipe installs custom wrapper scripts, usage notes should be added to extra -> notes in the meta.yaml.

Example Yaml for unicycler tool:

```
package:
  name: unicycler
  version: 0.3.0b

build:
  number: 0
  skip: True # [py27]

source:
  fn: unicycler_0.3.0b.tar.gz
  url: https://github.com/rrwick/Unicycler/archive/
↪906a3e7f314c7843bf0b4edf917593fc10baee4f.tar.gz
  md5: 5f06d2bd8ef5065c8047421db8c7895f

requirements:
  build:
  - python
  - setuptools
  - gcc

  run:
  - python
  - libgcc
  - spades >=3.6.2
  - pilon
  - java-jdk
  - bowtie2
  - samtools >=1.0
  - blast
  - freebayes

test:
  commands:
    - unicycler -h
```

(continues on next page)

(continued from previous page)

```
    - unicycler_align -h
    - unicycler_check -h
    - unicycler_polish -h

about:
  home: https://github.com/rrwick/Unicycler
  license: GPL-3.0
  license_file: LICENSE
  summary: 'Hybrid assembly pipeline for bacterial genomes'
```

When the recipe is ready a Pull Request should be created on the bioconda-recipes github repo. Finally the container is automatically created for the new BioConda Package if everything is corrected

The following are the detailed steps involved in creating bioconda based biocontainers:

2.1 - One-time Setup

2.1.1 - Install Bioconda

Bioconda is a channel for the conda package manager specializing in bioinformatics software. It consists of:

- A repository of recipes hosted on GitHub
- A build system that turns these recipes into conda packages
- A repository of >1500 bioinformatics packages ready to use with a simple conda install command

Over 130 contributors that add, modify, update and maintain the recipes

Important: **Recipe vs package** A **recipe** is a directory containing small set of files that defines name, version, dependencies, and URL for source code. A recipe typically contains a meta.yaml file that defines these settings and a build.sh script that builds the software. A recipe is converted into a package by running "conda-build" on the recipe. A **package** is a bgzipped tar file (.tar.bz2) that contains the built software. Packages are uploaded to anaconda.org so that users can install them with "conda install" command.

Bioconda requires the conda package manager to be installed. If you have an Anaconda Python installation, you already have it. Otherwise, the best way to install it is with the Miniconda package.

Warning: Bioconda supports only 64-bit Linux and Mac OSX. The Python 3 version is recommended.

Install miniconda specific for your platform. The following code shows the Miniconda installation on MacOSX and Linux

- MacOSX

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
bash Miniconda3-latest-MacOSX-x86_64.sh
```

- Linux

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

Accept all the default settings and let conda prepend the PATH in *~/.bashrc*

```
source ~/.bashrc
```

If you already have miniconda installed on your MacOSX/Linux, you can update that using

```
conda upgrade conda
conda upgrade conda-build
```

2.1.2 - Setting up of Channels of Bioconda

After installing conda you will need to add the Bioconda channel as well as the other channels Bioconda depends on. It is important to run the following commands in this order so that the priority is set correctly.

```
conda config --add channels conda-forge
conda config --add channels defaults
conda config --add channels r
conda config --add channels bioconda
```

2.1.3 - Test Bioconda installation

After installing Bioconda and setting-up channels, test to see if the installation of Bioconda worked properly by installing a package

```
conda install <package>

# Example
conda install bwa
# Or a specific version can be installed like this
conda install bwa=0.7.12
```

If there are no errors during installation of *bwa*, your Bioconda set-up is complete

2.1.4 - Next, create a fork of bioconda-recipes repo onto your GitHub account and then clone it locally.

Warning: Create a github account if you don't have one already.

```
git clone https://github.com/<githubUSERNAME>/bioconda-recipes.git
```

2.1.5 - Add the main bioconda-recipes repo as an upstream remote to more easily update your branch with the upstream master branch

```
cd bioconda-recipes
git remote add upstream https://github.com/bioconda/bioconda-recipes.git
```

2.1.6 - Request to be added to the Bioconda team

While not required, you can be added to the Bioconda by posting in Issue #1. Members of the Bioconda team can merge their own recipes once tests pass, though we ask that first-time contributions and anything out of the ordinary be reviewed by the @bioconda/core team.

Even if you are a member of the Bioconda team with push access, using your own fork will allow testing of your recipes on *travis-ci* using your own account's free resources without consuming resources allocated by travis-ci to the Bioconda group. This makes the tests go faster for everyone.

Create the Tool's Required Bioconda recipe (for generating Biocontainers)

2.1.7 - Update Bioconda repo and requirements

Before starting, it's best to update your fork with any changes made recently to the upstream Bioconda repo. Assuming you've set up your fork as above:

```
git checkout master
git pull upstream master
```

2.1.8 - Checkout a new branch

Check out a new branch in your fork (here the branch is arbitrarily named my-recipe):

```
git checkout -b my-recipe
```

2.1.9 - Create the recipe

Before you create a recipe, make sure to check that package exists for that recipe. If the package is already present, then you don't need to create the recipe. There are couple of ways to check for the package

1. Search for the package name in here
2. Search for the package name on the command line

```
conda search <package> -c bioconda

# Example
conda search taco -c bioconda
```

If the package of your interest, is not available, you can create the Bioconda recipe for the tool of your interest as below

```
conda skeleton <source> <package>
```

Note: Source: The source of the tool can be pypi, cran, bioconductor or cpan. Guidelines for Bioconda recipe Package: The name of the package

If the tool is not available from any of the above sources, then you need to generate a Bioconda package from scratch.

2.2.0 - Test it locally

After creating your recipe (using one of the above methods), to make sure your recipe works, you need to test it locally. There are two options.

2.2.1 - Quick test

The quickest, but not necessarily most complete, is to run *conda-build* command directly

```
conda install conda-build
conda build ./<package>

# Example
conda build bowtie2/2.2.4
```

2.2.2 - Push your changes to your fork on github repo

Once your tests are successful, before pushing your changes to your fork on github, it is best to merge any changes that have happened recently on the upstream master branch. See sycncing a fork for details, or run

```
git fetch upstream
```

syncs the fork's master branch with upstream

```
git checkout master
git merge upstream/master
```

merges those changes into the recipe's branch

```
git checkout my-recipe
git merge master
```

```
push your changes to your fork on github
git push origin my-recipe
```

2.2.3 - Open a pull request on the bioconda-recipes repo

Tip: If it's your first recipe or the recipe is doing something non-standard, please ask *@bioconda/core* for a review.

2.2.4 - Test the built bioconda package and Biocontainer

After the pull request, travis-ci will again do the builds to make sure everthing works. When the pull request is merged with the master branch by Bioconda team, the package will be uploaded to anaconda.org and Biocontainers will be pushed to quay.io.

2.2.5 - Testing the Bioconda package

Once the Bioconda package is available on Anaconda and biocontainer is available on *quay.io*, it may be a good idea to test those in staging first, so that production jobs aren't interrupted.

2.2.5.1 - Install the built Bioconda package from Anaconda (optional but recommended)

```
conda create -n myenvironment my-package # This is optional but it is always good to␣
↪test this
```

This method will install the package in the */home/username/minconda3/envs/myenvironment/bin*

2.2.5.2 - Testing the Biocontainer

- Pull your Biocontainer from quay.io of your new recipe (Mandatory)

```
docker pull quay.io/biocontainers/<my-package>:<version-number>--<python-version>_
↪<built-number>
```

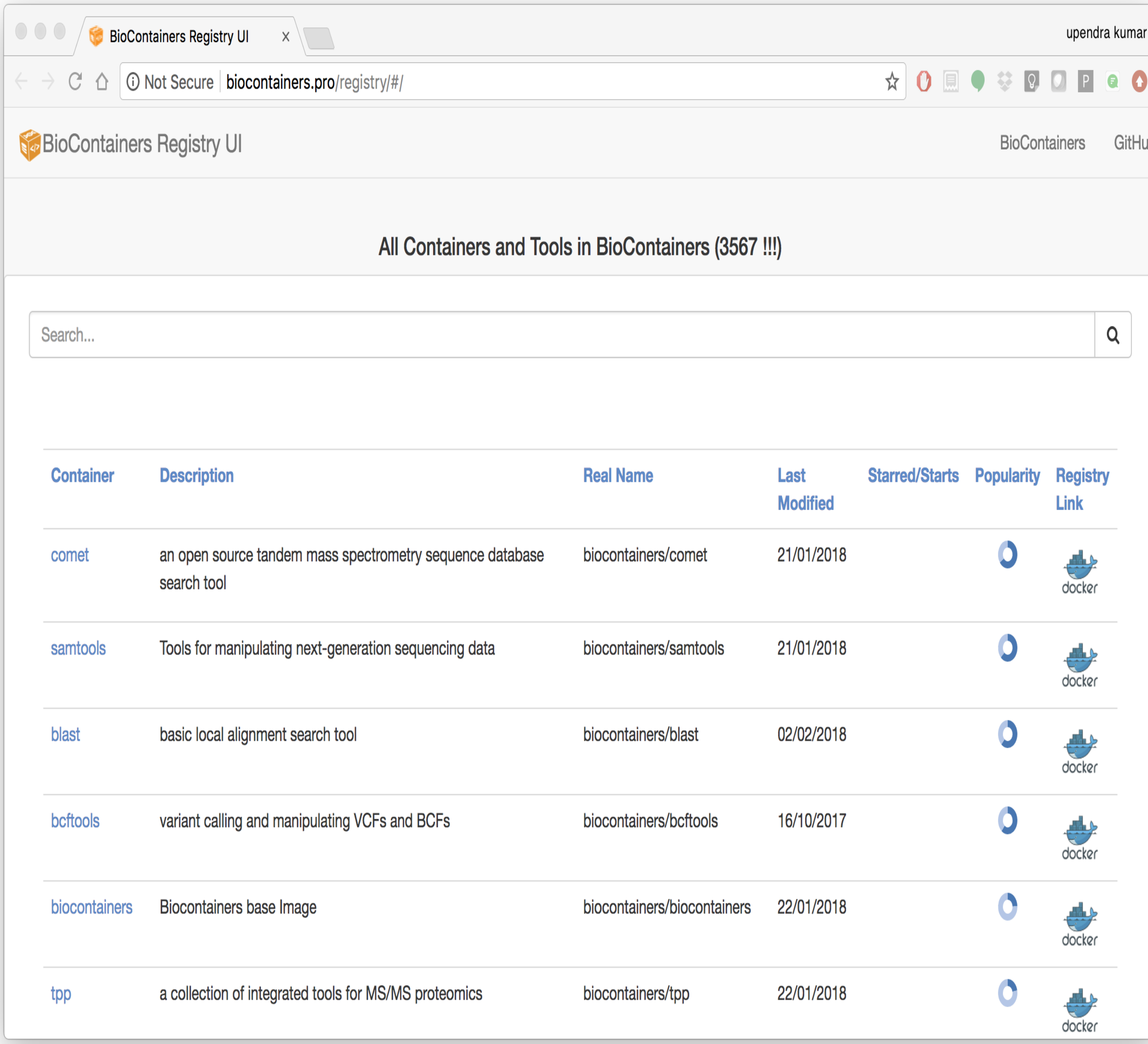
Run the tool's Biocontainer using the image:tag name, sample parameters, and inputs given in the tool request with a docker run command.

If the tool crashes, or the output does not match the sample output, contact the Bioconda or the user who creates the Biocontainers. Clean up any data containers and dangling images created in testing with *docker rm -v* and *docker rmi* commands. This command will cleanup any 'dangling' images:

```
docker rmi $(docker images -f 'dangling=true' -q)
```

18.2 The BioContainers Registry

BioContainers Registry UI provides the interface to search, tag, and document a BioContainers across all the registries.


BioContainers Registry UI
upendra kumar
Not Secure biocontainers.pro/registry/#/
BioContainers Registry UI
BioContainers
GitHu
Search...

All Containers and Tools in BioContainers (3567 !!!)

Container	Description	Real Name	Last Modified	Starred/Starts	Popularity	Registry Link
comet	an open source tandem mass spectrometry sequence database search tool	biocontainers/comet	21/01/2018			docker
samtools	Tools for manipulating next-generation sequencing data	biocontainers/samtools	21/01/2018			docker
blast	basic local alignment search tool	biocontainers/blast	02/02/2018			docker
bcftools	variant calling and manipulating VCFs and BCFs	biocontainers/bcftools	16/10/2017			docker
biocontainers	Biocontainers base Image	biocontainers/biocontainers	22/01/2018			docker
tpp	a collection of integrated tools for MS/MS proteomics	biocontainers/tpp	22/01/2018			docker

The users can search containers by using the search box

The containers registry allow the users to sort the containers by any of these properties:

- Container Name: Container Name
- Description: Description Provided by the developer of the container.
- Real Name: The corresponding registry + container name
- Last Modified: Last date where the container has been modified.
- Starred/Start: If the container has been starred in any of the repos.

- Popularity: How many times a container has been pull from a registry.
- Registry Link: the registry Link.

CHAPTER 19

Biocontainers in HPC

On HPC systems, the traditional way to make software available is through the "modules" system. The modules system allows administrators to install hundreds of scientific software packages on a system, including multiple versions of the same package, and then users can select the packages they want loaded into their environment.

Adding a package into the modules system can take a lot of work. At TACC, the process of adding a module looks something like this:

1. Manually install the software on the system as a test to figure out the best compiler optimizations, account for any dependencies, etc.
2. Encode the installation process into an RPM file, add relevant metadata to the RPM, and craft a "modulefile" that lets the modules system discover the app.
3. Build the RPM and put the compiled package in a designated location.
4. Send a ticket to the administrators so that they can install the package during the next maintenance.
5. System administrators install the package through a scripted process that installs the package on the local hard-drive on every node on the system (often thousands of nodes)
6. Next time users look for the package using the "modules" command, it will show up and be available for use.

Note: New systems are usually different enough from the old systems that RPM files must be altered manually every time, for every package. TACC usually has 6-10 production systems that need scientific software RPMs. For examples of what goes into an RPM file, you can look in this Github repository

19.1 Installing thousands of apps on a cluster

As the BioContainers project has demonstrated, the Bioinformatics community alone uses thousands of software packages. What can users of HPC systems do to gain access to the software they care about when the modules system cannot keep up?

Bjorn Gruning from the University of Freiburg, along with many many collaborators from the BioContainers community has championed the inclusion of Singularity images from the BioContainers project. This means that every app with a Conda recipe also has a Singularity image for it. They are publicly available via FTP here: https://depot.galaxyproject.org/singularity/

They are also already available on every TACC system on the global work filesystem in this directory:

```
/work/projects/singularity/TACC/biocontainers/
```

There are **over 7200** Singularity images currently available in that directory. (So be careful trying to do an "ls" command there!

All the images are named in the format: name_version.img. They were converted from Docker containers using the docker2singularity script that adds top level directories (like "/work") to make sure that volume mounts will work on the HPC systems. If you use TACC, you can check for a software package, for example, bwa, like this:

```
$ find /work/projects/singularity/TACC/biocontainers/ -name bwa*
/work/projects/singularity/TACC/biocontainers/bwameth_0.20--py35_0.img
/work/projects/singularity/TACC/biocontainers/bwa_0.5.9--0.img
/work/projects/singularity/TACC/biocontainers/bwa_0.7.17--pl5.22.0_0.img
/work/projects/singularity/TACC/biocontainers/bwameth_0.2.0--py36_1.img
/work/projects/singularity/TACC/biocontainers/bwameth_0.2.0--py36_0.img
/work/projects/singularity/TACC/biocontainers/bwa_0.7.13--1.img
/work/projects/singularity/TACC/biocontainers/bwa_0.7.15--1.img
/work/projects/singularity/TACC/biocontainers/bwa_0.7.16--pl5.22.0_0.img
```

To test the image interactively (from a compute node!) you can do something like the following:

```
# get an interactive session
idev
# load the Singularity module
module load tacc-singularity
# explore the container
singularity shell /work/projects/singularity/TACC/biocontainers/bwa_0.7.15--1.img
```

Note: If you run the container from your $WORK directory, Singularity will by default volume mount /work into the container, and all your $WORK files will be available.

From there, you are ready to incorporate "singularity exec" commands into your job submission scripts just like you would any other command.

CHAPTER 20

Docker related resources

Awesome Docker

Docker labs

Docker Community Slack

Docker Community Forums

Docker hub

Docker documentation

Docker on StackOverflow

Docker on Twitter

Play With Docker Hands-On Labs

Docker tips

Docker cloud

Docker store

Interesting tutorials and blog posts:

1. Docker Blog
2. A beginner friendly intro to VMs and Docker
3. Intro to Docker from Neurohackweek
4. Understanding Images

CHAPTER 21

Singularity related resources

Singularity Homepage

Singularity Hub

University of Arizona Singularity Tutorials

NIH HPC

Dolmades - Windows Apps in Linux Docker-Singularity Containers *Warning not tested*

21.1 Singularity Talks

Gregory Kurtzer, creator of Singularity has provided two good talks online: Introduction to Singularity, and Advanced Singularity.

Vanessa Sochat, lead developer of Singularity Hub, also has given a great talk on Singularity which you can see online.

CHAPTER 22

Other resources

University of Arizona Campus Resources

- UA Campus Accessibility
- UA Campus Transportation
- Family Spaces and Lactation Support
- BIO5 Institute
- Transportation beyond BIO5 and UA campus

CHAPTER 23

For instructors!

Coordinating Web site work

Please create a pull request as soon as you start editing something, rather than waiting! That way you can tell others what you're working on.

You could/should also mention it on Slack in the "cc-leads" channel.

Technical info re adding content to the Web site

All the Container Camp workshop tutorials are stored on GitHub.

We will use GitHub Flow for updates: from the command line,

- fork container camp repository;
- edit, change, add, etc;
- submit a PR;
- when ready to review & merge say 'ready for review & merge @cc2018'.

It's important that all updates go through code review by someone. Anyone with push access to the repo can review and merge!

From the Web site, you should be able to edit the files and then set up a PR directly. You can also fork the repo, perform multiple edits and submit a PR through the web interface.

Updating the "official" Web site.

The Web site, will update automatically from GitHub. However, it may take 5-15 minutes to do so.

Building a local copy of the Web site.

Briefly,

- clone the repo:

  ```
  git clone https://github.com/CyVerse-learning-materials/
  container_camp_workshop_2018.git
  ```

- set up a virtualenv with python2 or python3:

  ```
  python -m virtualenv buildenv -p python3.5; . ~/buildenv/bin/activate
  ```

- install the prerequisites:

  ```
  pip install -r requirements.txt
  ```

- build site:

  ```
  make html
  ```

- open / click on

  ```
  _build/html/index.html
  ```

Formatting, guidelines, etc.

Everything can/should be in Restructured text If you're not super familiar with Restructured text, you can use online restructured text editor to write your tutorials.

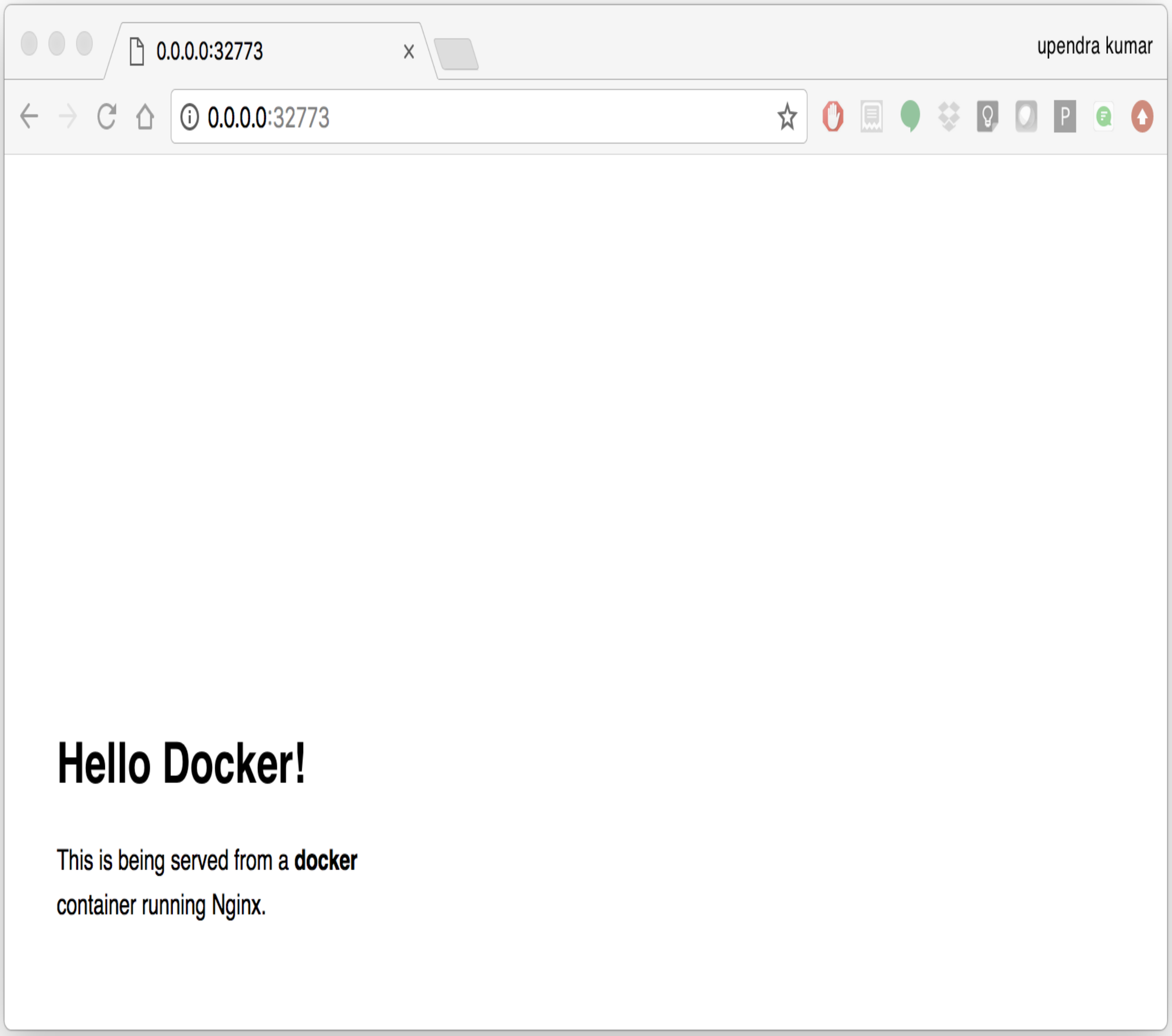
(Note that you can go visit the github repo and it will helpfully render *.rst* files for you if you click on them! They just won't have the full site template.)

Files and images that don't need to be "compiled" and should just be served up through the web site can be put in the *_static* directory; their URL will then be

> https://cyverse-container-camp-workshop-2018.readthedocs-hosted.com/_static/filename

Images

Image formatting in Restructured text is pretty straightforward. Here is an example



Hello Docker!

This is being served from a **docker** container running Nginx.

.. |*static_site_docker*| *image:: ../img/static_site_docker.png*

:width: 750

:height: 700

CHAPTER 24

Problems? Bugs? Questions?

- If there is a bug and you can fix it: submit a PR. Make sure that I know who you are so that I can thank you.
- If there is a bug and you can't fix it, but you can reproduce it: submit an issue explaining how to reproduce.
- If there is a bug and you can't even reproduce it: sorry. It is probably an Heisenbug. We can't act on it until it's reproducible, alas.
- If you have attended this workshop and have feedback, or if you want somebody to deliver that workshop at your conference or for your company: you can contact one of us!

upendra at cyverse dot org

Thank you!