COE 332: Software Engineering & Design

Texas Advanced Computing Center

Feb 11, 2022

COURSE SCHEDULE:

The objective of this course is to introduce students to advanced computing concepts in software engineering, software systems design, cloud computing and distributed systems, and computational engineering. Through a series of assignments spanning the course of the semester, students will build a cloud-based, computational system to analyze a time series dataset and provide a web-accessible interface to their system.

CHAPTER

ONE

WEEK 1: ONBOARDING, LINUX, PYTHON REVIEW

We will use the first week of class to set expectations and describe requirements for the upcoming semester. We will begin to set up some necessary accounts and onboard students to the TACC / class ecosystem. We will also quickly review essential Linux and Python skills that will be necessary for the rest of this course.

1.1 Class Introduction

Welcome to **COE 332: Software Engineering & Design!**

Course Time: Every T/TH 11:00am - 12:30pm on Zoom

The objective of this course is to introduce students to advanced computing concepts in software engineering, software systems design, cloud computing and distributed systems, and computational engineering. Through a series of assignments spanning the course of the semester, students will build a cloud-based, computational system to analyze a time series dataset and provide a web-accessible interface to their system.

Note: Zoom recordings should be automatically uploaded to Canvas following each lecture.

1.1.1 About the Instructors

Joe Allen

Joe Allen joined the TACC Life Sciences Computing Group in 2015 where he works to advance computational biology and bioinformatics research at UT system academic and health institutions. His research experience spans a range of disciplines from computer-aided drug design to wet lab biochemistry. He is also interested in exploring new ways high performance computing resources can be used to answer challenging biological questions. Before joining TACC, Joe earned a B.S. in Chemistry from the University of Jamestown (2006), and a Ph.D. in Biochemistry from Virginia Tech (2011).

Charlie Dey

Charlie is the Director of Training and Professional Development with the User Services group at TACC with a background in web development and scientific computing. Charlie's responsibilities at TACC include organizing, developing content, and building curriculums for TACC's academic course selection taught in conjunction with several departments at the University of Texas at Austin, as well as for TACC's professional development and educational training. Prior to joining TACC, he worked as a Senior Application Developer for the Carle Foundation, and as a computer science instructor at Parkland College in Champaign, IL. He was also a member of a specialized application development team at the University of Illinois and has also been a contracted research consultant for NASA Ames Research Center, studying computational immunology and bioinformatics. Charlie holds a Bachelor's Degree concentrating in Computer Science and Biology from Eastern Illinois University, and certifications in 3D programing and visualization.

Brandi Kuritz

Brand joined TACC in 2018 as an intern and established CIC's support infrastructure while learning software engineering and computer science fundamentals. She is now employed full time as a software developer and is currently working on development for TACC APIs and supporting utilities.

Joe Stubbs

Joe leads the Cloud and Interactive Computing (CIC) group, which focuses on building cloud native applications and infrastructure for computational science. CIC develops, deploys and maintains multiple national-scale clouds as part of ongoing projects funded by the National Science Foundation. Additionally, the CIC group contributes to and deploys multiple cloud platforms-as-a-service for computational science including the Agave science-as-a-service platform, TACC's custom JupyterHub, and Abaco: Functions-as-a-service via the Actor model and Linux containers. These platforms are leveraged by numerous cyberinfrastructure projects used by tens of thousands of investigators across various domains of science and engineering. Prior to joining the University of Texas, Joe received a B.S. in Mathematics from the University of Texas, Austin and a Ph.D. in Mathematics from the University of Michigan. His recent interests include distributed systems, container technologies and interactive scientific computing.

1.1.2 About the Syllabus

Grades for the course will be based on the following:

- 30% Homework - Approximately 8 assignments, assigned on Tuesdays and due the following Tuesday. The lowest score will be dropped.
- 30% Midterm - We will have a written midterm.
- 40% Project - Students will form groups and will submit a final class project consisting of a distributed, web-accessible, cloud system to analyze a time series dataset. The project will draw from and build upon work done throughout the semester in homework assignments. More details will be given in the upcoming weeks.

Approximate Schedule and Key Dates (subject to change):

- Week 1: Intro, Linux, Python Review
- Week 2: JSON, Unit Testing
- Week 3: Version Control, Intro to Containers
- Week 4: Advanced Containers, YAML, Docker Compose
- Week 5: HTTP, REST, Intro to Flask
- Week 6: Advanced Flask, Integration Testing
- Week 7: Databases, Persistence in REST
- Week 8: Review, Midterm
- Spring Break
- Week 9: Virtualization: Container Orchestration and Kubernetes
- Week 10: Virtualization: Container Orchestration and Kubernetes Cont
- Week 11: Continuous Integration
- Week 12: Asynchronous Programming
- Week 13: Queues
- Week 14: Special Topics
- Week 15: Special Topics - Final Week of Class

- Last Day of Class: Final Project Due

1.1.3 Office Hours

Office hours will be held every T/TH for 1 hour immediately following the class. Office hours will also be available by appointment. No in-person office visits are available this semester. Instead, we will meet by Zoom (*ad hoc*) and/or in the dedicated class Slack space:

https://tacc-learn.slack.com/

Within that space, you will find two channels dedicated to this class: *#coe332-general* and *#coe332-officehours*. The general channel will be used for general discussion, posting links and materials, and making general announcements to the class. The office hours channel will be used to interact with course instructors and receive help. **Students are expected be signed in and checking slack at least twice a week.**

> **Attention:** Everyone please click on the Slack link above and request access

1.1.4 Key Prerequisites

This course assumes familiarity with the Python programming language and strong working knowledge of basic, high-level language programming concepts including: data structures, loops and flow control, and functions. We also assume a basic, working knowledge of the Linux command line.

We will briefly review programming concepts in Linux and Python during the first week of class, the first homework assignment will be based on these topics, and we will make every effort to help students who are less familiar with these concepts in Python. Ultimately, each student is expected to and responsible for mastering this material. This is not an introductory programming class and we will not have time to give a comprehensive treatment of all of these topics.

You will need an SSH client and way to edit / run Python code to be successful in this class. There are many programs available, and it does not matter much which you choose as long as you are comfortable using them.

SSH Client:

- PuTTY (Windows)
- MobaXterm (Windows)
- Windows Subsystem for Linux (Windows)
- Terminal (Mac, Linux)
- iTerm (Mac)

Python IDE

- Terminal + VIM or Emacs or Nano (Mac, Linux)
- VSCode (Windows, Mac, Linux)
- Atom (Windows, Mac, Linux)
- PyCharm (Windows, Mac, Linux)

1.1.5 Additional Help

Our main goal for this class is your success. Please contact us if you need extra help.

Joe Allen - wallen [at] tacc [dot] utexas [dot] edu

Charlie Dey - charlie [at] tacc [dot] utexas [dot] edu

Brandi Kuritz - bkuritz [at] tacc [dot] utexas [dot] edu

Joe Stubbs - jstubbs [at] tacc [dot] utexas [dot] edu

1.2 Onboarding to TACC

The Texas Advanced Computing Center (TACC) at UT Austin designs and operates some of the world's most powerful computing resources. The center's mission is to enable discoveries that advance science and society through the application of advanced computing technologies.

We will be using cloud resources at TACC as our development environment. We will access the cloud resources via our SSH clients and TACC account credentials.

Attention: Everyone please apply for a TACC account now using this link. If you already have a TACC account, you can just use that. Send your TACC username to the course instructors via Slack or e-mail as soon as possible (see below).

```
To: {wallen, charlie, bkuritz, jstubbs} [at] tacc [dot] utexas [dot] edu
From: you
Subject: COE 332 TACC Account
Body: Please include your name, EID, TACC user name
```

1.2.1 About TACC

TACC is a Research Center, part of UT Austin, and located at the JJ Pickle Research Campus.

TACC at a Glance

Other TACC Services

- Portals and gateways
- Web service APIs
- Rich software stacks
- Consulting
- Curation and analysis
- Code optimization
- Training and outreach
- => Learn more

TACC Partnerships

- NSF: Leadership Class Computing Facility (LCCF)
- NSF: Extreme Science and Engineering Discovery Environment (XSEDE)

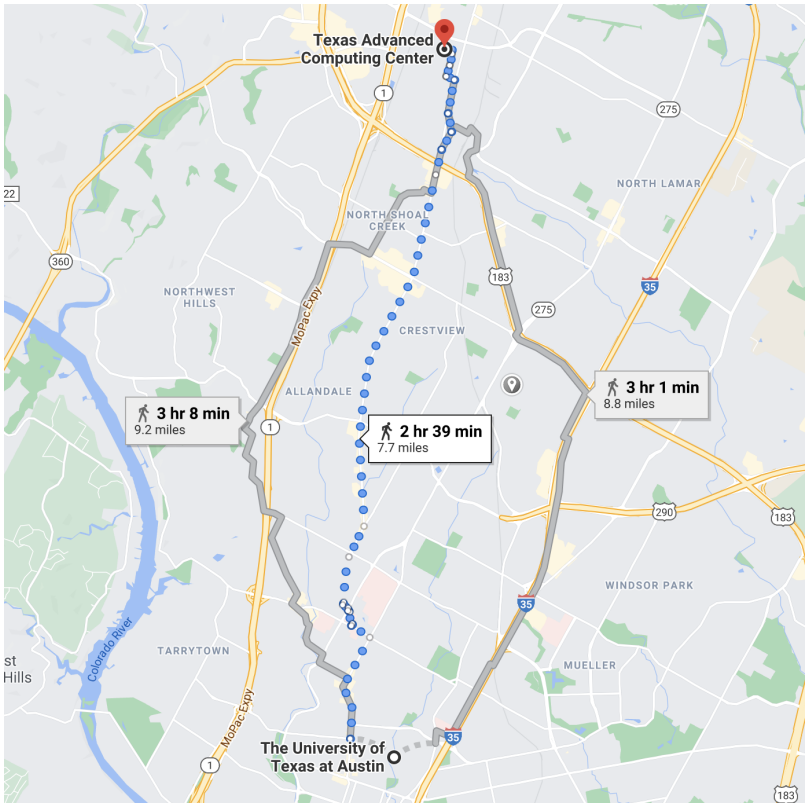
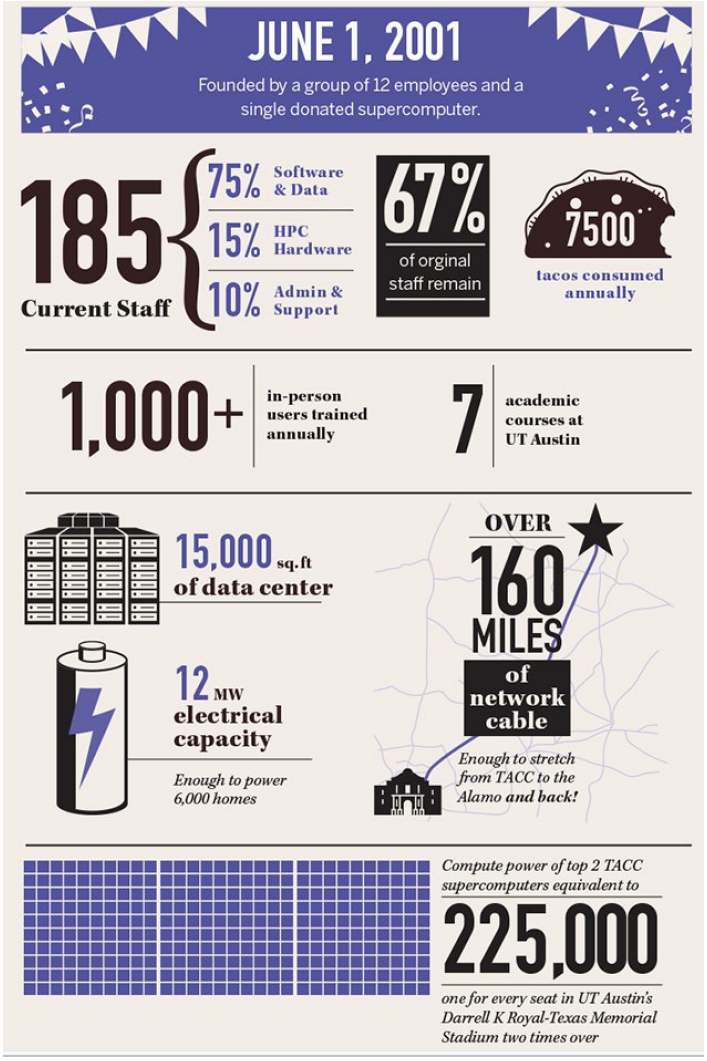
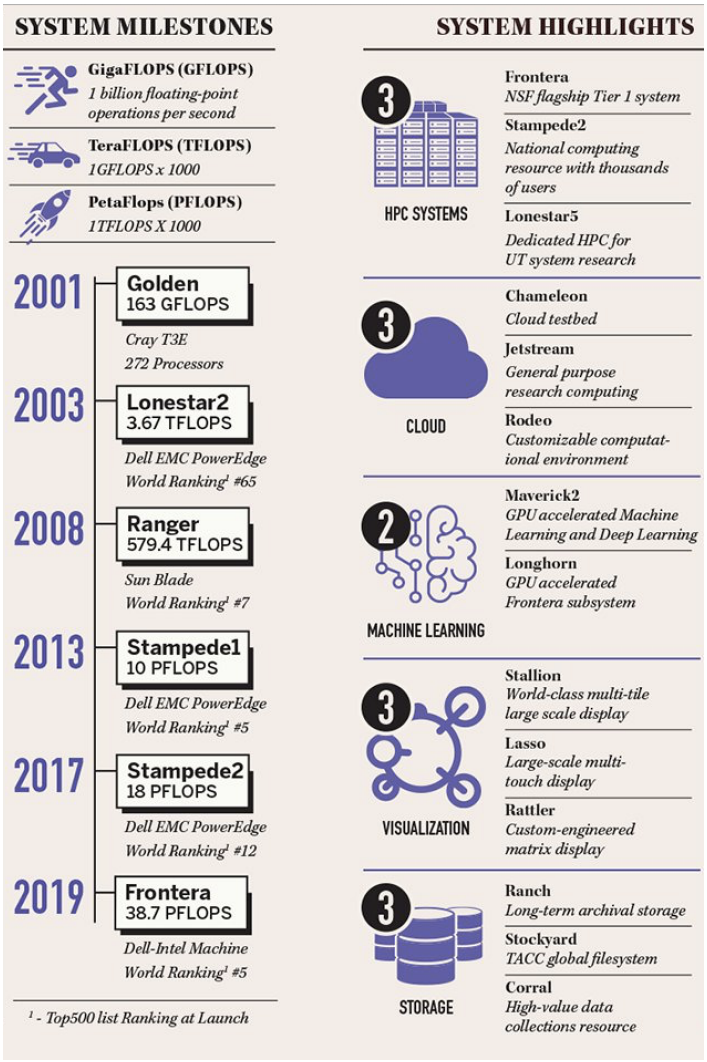


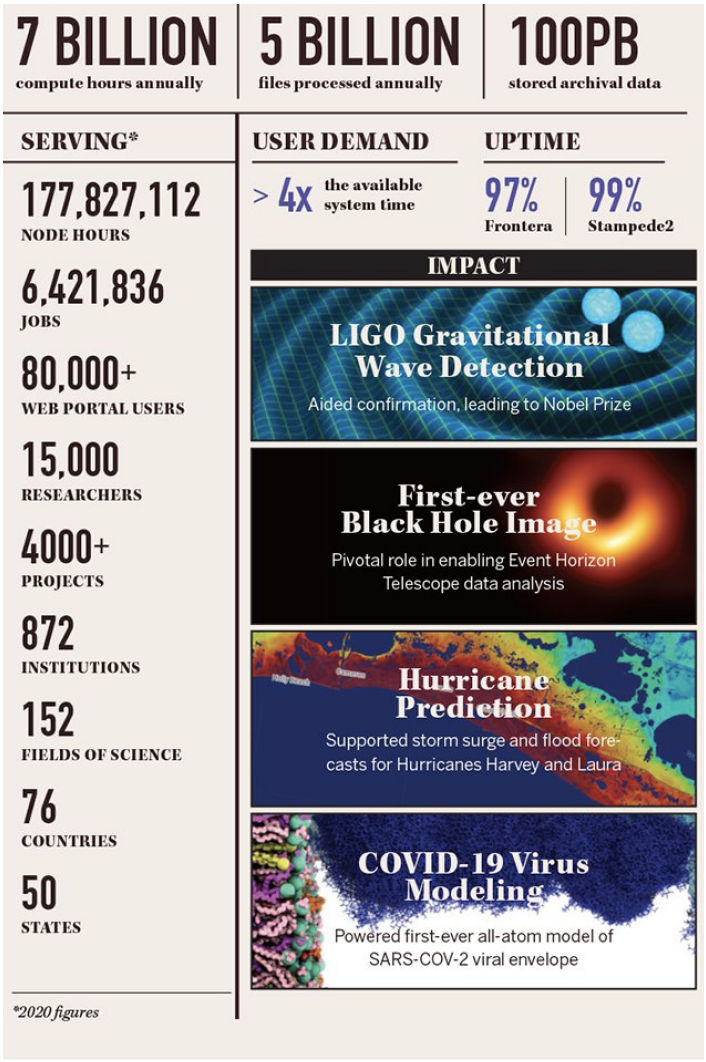
Fig. 1: A short 7.7 mile walk from main campus!

Fig. 2: One of two TACC buildings located at JJ Pickle.

Fig. 3: A tall guy standing among taller Frontera racks.




SYSTEM MILESTONES
GigaFLOPS (GFLOPS)
1 billion floating-point operations per second
TeraFLOPS (TFLOPS)
1GFLOPS x 1000
PetaFlops (PFLOPS)
1TFLOPS X 1000
2001
Golden
163 GFLOPS
Cray T3E
272 Processors
2003
Lonestar2
3.67 TFLOPS
Dell EMC PowerEdge
World Ranking[1] #65
2008
Ranger
579.4 TFLOPS
Sun Blade
World Ranking[1] #7
2013
Stampede1
10 PFLOPS
Dell EMC PowerEdge
World Ranking[1] #5
2017
Stampede2
18 PFLOPS
Dell EMC PowerEdge
World Ranking[1] #12
2019
Frontera
38.7 PFLOPS
Dell-Intel Machine
World Ranking[1] #5
[1] - Top500 list Ranking at Launch
SYSTEM HIGHLIGHTS
3
HPC SYSTEMS
Frontera
NSF flagship Tier 1 system
Stampede2
National computing resource with thousands of users
Lonestar5
Dedicated HPC for UT system research
3
CLOUD
Chameleon
Cloud testbed
Jetstream
General purpose research computing
Rodeo
Customizable computational environment
2
MACHINE LEARNING
Maverick2
GPU accelerated Machine Learning and Deep Learning
Longhorn
GPU accelerated Frontera subsystem
3
VISUALIZATION
Stallion
World-class multi-tile large scale display
Lasso
Large-scale multi-touch display
Rattler
Custom-engineered matrix display
3
STORAGE
Ranch
Long-term archival storage
Stockyard
TACC global filesystem
Corral
High-value data collections resource


7 BILLION
compute hours annually
5 BILLION
files processed annually
100PB
stored archival data
SERVING*
177,827,112
NODE HOURS
6,421,836
JOBS
80,000+
WEB PORTAL USERS
15,000
RESEARCHERS
4000+
PROJECTS
872
INSTITUTIONS
152
FIELDS OF SCIENCE
76
COUNTRIES
50
STATES
*2020 figures
USER DEMAND
> 4x the available system time
UPTIME
97% Frontera
99% Stampede2
IMPACT
LIGO Gravitational Wave Detection
Aided confirmation, leading to Nobel Prize
First-ever Black Hole Image
Pivotal role in enabling Event Horizon Telescope data analysis
Hurricane Prediction
Supported storm surge and flood forecasts for Hurricanes Harvey and Laura
COVID-19 Virus Modeling
Powered first-ever all-atom model of SARS-COV-2 viral envelope

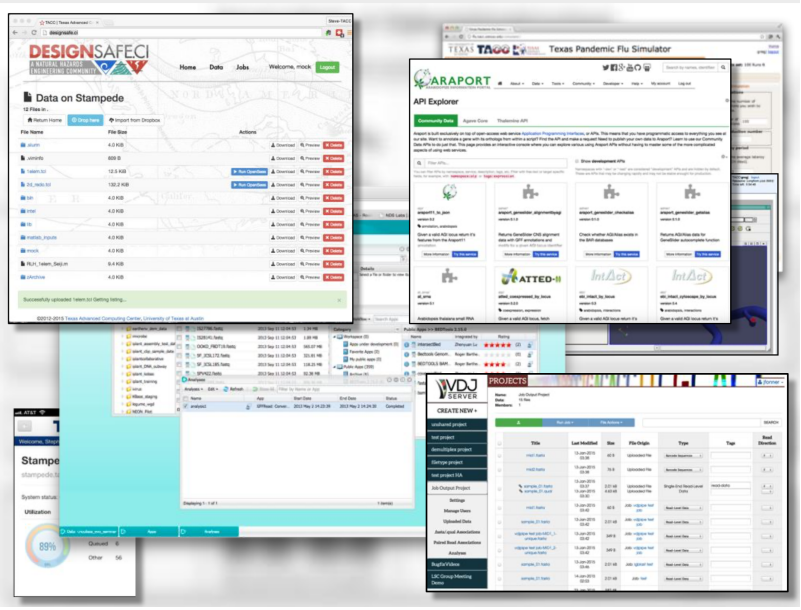


Fig. 4: Snapshot of a few of TACC's portal projects.

- UT Research Cyberinfrastructure (UTRC)
- TX Lonestar Education and Research Network (LEARN)
- Industry, STAR Program
- International, The International Collaboratory for Emerging Technologies
- => Learn more

Attention: Did you already e-mail your TACC username to the course instructors?

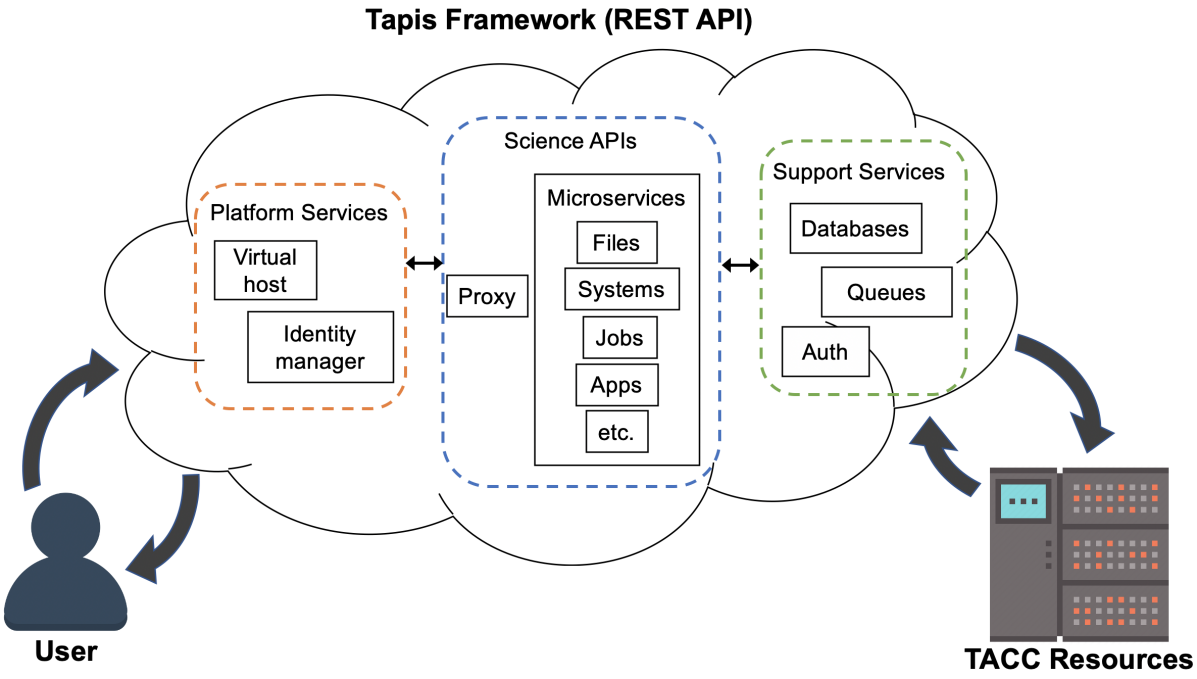
Which brings us to the question of why are we here teaching this class?

1.2.2 Engineering Complex Systems in the Cloud

The Tapis Framework, developed at TACC, is a great example of a complex assembly of code with many moving parts, engineered to help researchers interact with high performance computing systems in streamlined and automated ways. Tapis empowers its users to:

- Authenticate using TACC (or other) credentials
- Manage, move, share, and publish data sets
- Run scientific code in batch jobs on clusters
- Set up event-driven processes
- Many other things!

The above description of Tapis and the below schematic diagram are both intentionally left a little bit vague as we will cover more of the specifics of Tapis later on in the semester.



Tip: Astute observers may notice that most, if not all, tools, technologies, and concepts that form the Tapis ecosystem show up somewhere in the agenda for COE 332.

1.2.3 Demo Applications of Tapis

So what can you do with Tapis?

Why would I want to build something similar?

Why should I learn how to use all of these tools and technologies?

Without concrete examples, it can seem rather esoteric. The two vignettes below hopefully illustrate how a carefully designed framework can be employed to tackle real-world problems.

Vignette 1: Drug Discovery Portal

Problem: The early stages of drug discovery employ a computational process called "virtual screening" to narrow millions or even billions of potential drug hits down to a few hundred or thousand that can be tested in a lab. The virtual screening process can be computationally intensive and difficult for novice users to do well.

Importance: Virtual screening can save a lot of time and money in the drug discovery process by narrowing the search. Small molecules can be tested for compatibility with protein targets before the wet lab research begins.

Approach: Faculty and staff from UTMB Galveston and TACC used the Tapis framework to deploy a service for virtual screening in a point-and click web interface.

Result: Users of the "Drug Discovery Portal" can upload target proteins and click a few buttons to start running large-scale virtual screens directly on TACC clusters. No prior experience in virtual screening, the Linux command line interface, or batch queueing systems is required.

Source: https://doi.org/10.1021/ci500531r

Vignette 2: Real-Time Quantitative MRI

Fig. 5: Drug Discovery Portal web interface.

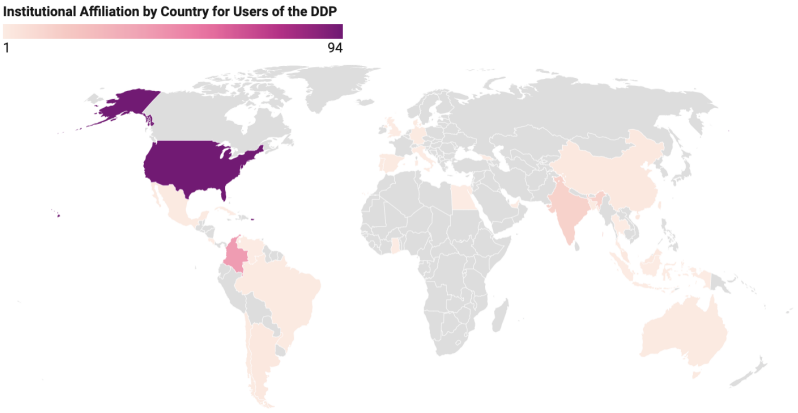


Fig. 6: Researchers from around the world using the platform.

Problem: Quantitative analysis of MR images is typically performed after the patient has left the scanner. Corrupted or poor quality images can result in patient call backs, delaying disease intervention.

Importance: Real-time analytics of MRI scans can enable same-session quality control, reducing patient call backs, and it can enable precision medicine.

Approach: Faculty and staff from UTHealth - Houston and TACC used the Tapis framework to help develop an automated platform for real-time MRI.

Result: Scan data can now be automatically processed on high performance computing resources in real-time with no human intervention.

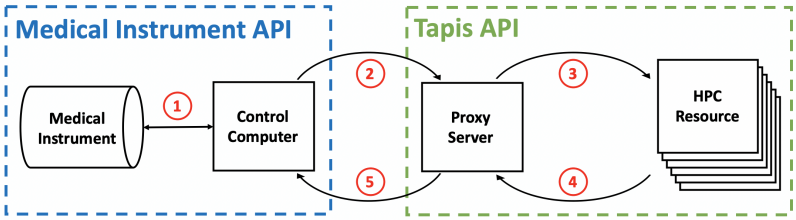


Fig. 7: Diagram of computer systems and APIs employed.

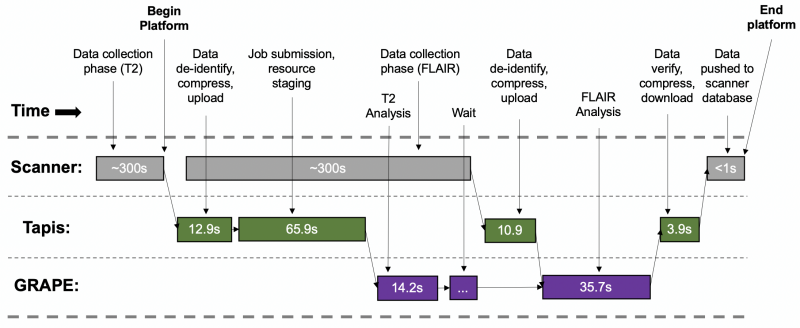


Fig. 8: Sample platform workflow for combining two images into one enhanced image.

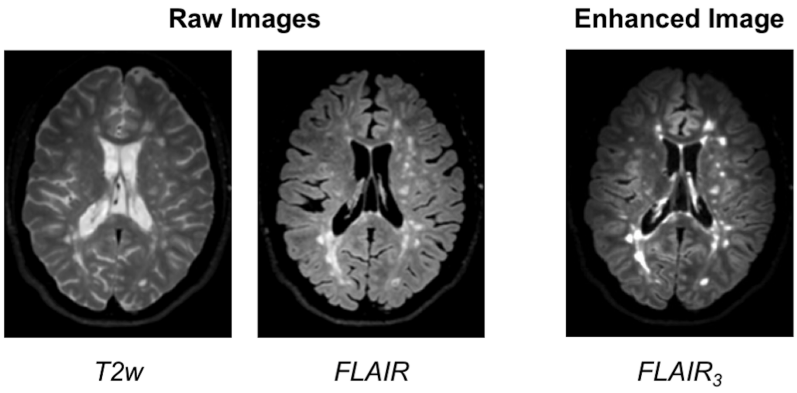


Fig. 9: Final image shows enhanced MS lesions.

Source: https://dx.doi.org/10.1109/JBHI.2017.2771299

Attention: If you already e-mailed your TACC account to the instructors, please go ahead and try the exercise below.

1.2.4 Bringing it All Together

Hopefully these examples start to show you what kind of software projects we will be working on this semester. Each week will be introducing a new concept, tool, or technology that will slowly be building to a larger overall framework with many moving parts.

1.2.5 For Next Time

Using your SSH client, please try to log in to the class server **before the next class period**:

```
[local]$ ssh username@isp02.tacc.utexas.edu
username@isp02.tacc.utexas.edu's password:
Last login: Sun Jan 17 23:48:54 2021 from cpe-24-27-53-74.austin.res.rr.com
------------------------------------------------------------------------
Welcome to the Texas Advanced Computing Center
   at The University of Texas at Austin

** Unauthorized use/access is prohibited. **

If you log on to this computer system, you acknowledge your awareness
of and concurrence with the UT Austin Acceptable Use Policy. The
University will prosecute violators to the full extent of the law.

TACC Usage Policies:
http://www.tacc.utexas.edu/user-services/usage-policies/

TACC Support:
https://portal.tacc.utexas.edu/tacc-consulting

------------------------------------------------------------------------
Intel(R) Parallel Studio XE 2017 Update 1 for Linux*
Copyright (C) 2009-2016 Intel Corporation. All rights reserved.
[remote]$ hostname -f
isp02.tacc.utexas.edu      # success!
```

1.3 Student Goals

When asked what their goals were for the course, students responded:

Python / Programming

- Become more familiar with utilizing Python
- get more programming experience
- to gain programming experience to be ready for the industry
- Get more practice and knowledge about programing
- have good programming project experience and figure out python

- Improve upon my programming skills and learn Python
- Gain experience in using python language for software development
- Improve my programming and learn Python
- Expand my programming knowledge for personal projects and prepare myself for the job market
- Improve my programming and algorithm skills

Python for Data Science

- I'm hoping to create an in depth project that utilizes python in hopes of using it to go into data science
- get more familiar with python and applying that to real life situations that could be helpful for the future
- good programming experience. I want to go into cancer research, so i'd like to learn how to use TACC's machines to handle big data

Technologies

- I want to learn Docker, databases, and more about APIs to be well equipped for software engineering jobs. Also this is a degree requirement for me
- k8s
- Get familiar with building web apps
- Learn about relevant computational methods.
- Gain a better understanding of cloud computing technologies
- Be more familiar with cloud computing and gain more experience in python to be prepared for the industry
- I want to learn more programming skills, specifically Kubernetes, and work on projects that can contribute to a portfolio!
- learn how to effectively use workflow resources to better my coding skills
- Learn more about cloud computing and improve software dev knowledge
- To broaden my skills in programming, specifically in cloud computing and workflow management in general
- I have previous experience using virtualized systems both locally (XCP-NG) and on AWS, but have not done much with containers. I am especially interested in learning about docker and k8s!

Software Engineering

- Build a complex computing system
- Be able to work in software engineering
- Understand what I did to finish the final project and be able to apply it
- I want to learn more about software engineering rather than just computation methods in order to create in depth projects
- Be able to build a complete software application the proper way
- learning more about programming applications
- learn how different languages connect to create a final product
- To understand how to create useful and larger scale software

This Person Gets It

- Most of my degree has been overly academic and not particularly practical; this class seems like it will provide something more that I desperately need.

Good news: You are all in the right place.

1.4 Linux Essentials

To be successful in this class, students should be able to:

- Describe basic functions of essential Linux commands
- Use Linux commands to navigate a file system and manipulate files
- Transfer data to / from a remote Linux file system
- Edit files directly on a Linux system using a command line utility (e.g. vim, nano, emacs)

Topics covered in this module include:

- Creating and navigating folders (`pwd`, `ls`, `mkdir`, `cd`, `rmdir`)
- Creating and manipulating files (`touch`, `rm`, `mv`, `cp`)
- Looking at the contents of files (`cat`, `more`, `less`, `head`, `tail`, `grep`)
- Network and file transfers (`hostname`, `whoami`, `logout`, `ssh`, `scp`, `rsync`)
- Text editing with vim (insert mode, normal mode, navigating, saving, quitting)

1.4.1 Log in to the Class Server

To log in to `isp02.tacc.utexas.edu`, follow the instructions for your operating system or ssh client below.

Mac / Linux

```
Open the application 'Terminal'
ssh username@isp02.tacc.utexas.edu
(enter password)
```

Windows

```
Open the application 'PuTTY'
enter Host Name: isp02.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
```

If you can't access the class server yet, a local or web-based Linux environment will work for this guide. However, you will need to access the class server for future lectures.

Try this Linux environment in a browser.

1.4.2 Creating and Navigating Folders

On a Windows or Mac desktop, our present location determines what files and folders we can access. I can "see" my present location visually with the help of the graphic interface - I could be looking at my Desktop, or the contents of a folder, for example. In a Linux command-line interface, we lack the same visual queues to tell us what our location is. Instead, we use a command - `pwd` (print working directory) - to tell us our present location. Try executing this command in the terminal:

```
$ pwd
/home/wallen
```

This home location on the Linux filesystem is unique for each user, and it is roughly analogous to C:\Users\username on Windows, or /Users/username on Mac.

To see what files and folders are available at this location, use the `ls` (list) command:

```
$ ls
```

I have no files or folders in my home directory yet, so I do not get a response. We can create some folders using the `mkdir` (make directory) command. The words 'folder' and 'directory' are interchangeable:

```
$ mkdir folder1
$ mkdir folder2
$ mkdir folder3
```

```
$ ls
folder1 folder2 folder3
```

Now we have some folders to work with. To "open" a folder, navigate into that folder using the `cd` (change directory) command. This process is analogous to double-clicking a folder on Windows or Mac:

```
$ pwd
/home/wallen/
$ cd folder1
$ pwd
/home/wallen/folder1
```

Now that we are inside `folder1`, make a few sub-folders:

```
$ mkdir subfolderA
$ mkdir subfolderB
$ mkdir subfolderC
$ ls
subfolderA subfolderB subfolderC
```

Use `cd` to Navigate into `subfolderA`, then use `ls` to list the contents. What do you expect to see?

```
$ cd subfolderA
$ pwd
/home/wallen/folder1/subfolderA
$ ls
```

There is nothing there because we have not made anything yet. Next, we will navigate back to the home directory. So far we have seen how to navigate "down" into folders, but how do we navigate back "up" to the parent folder? There are different ways to do it. For example, we could specify the complete path of where we want to go:

```
$ pwd
/home/wallen/folder1/subfolderA
$ cd /home/wallen/folder1
$ pwd
/home/wallen/folder1/
```

Or, we could use a shortcut, `..`, which refers to the **parent folder** - one level higher than the present location:

```
$ pwd
/home/wallen/folder1
$ cd ..
$ pwd
/home/wallen
```

We are back in our home directory. Finally, use the `rmdir` (remove directory) command to remove folders. This will not work on folders that have any contents (more on this later):

```
$ mkdir junkfolder
$ ls
folder1 folder2 folder3 junkfolder
$ rmdir junkfolder
$ ls
folder1 folder2 folder3
```

Before we move on, let's remove the directories we have made, using `rm -r` to remove our parent folder `folder1` and its subfolders. The `-r` command line option recursively removes subfolders and files located "down" the parent directory. `-r` is required for non-empty folders.

```
$ rm -r folder1
$ ls
folder2 folder3
```

Which command should we use to remove `folder2` and `folder3`?

```
$ rmdir folder2
$ rmdir folder3
$ ls
```

1.4.3 Creating and Manipulating Files

We have seen how to navigate around the filesystem and perform operations with folders. But, what about files? Just like on Windows or Mac, we can easily create new files, copy files, rename files, and move files to different locations. First, we will navigate to the home directory and create a few new folders and files with the `mkdir` and `touch` commands:

```
$ cd     # cd on an empty line will automatically take you back to the home directory
$ pwd
/home/wallen
$ mkdir folder1
$ mkdir folder2
$ mkdir folder3
$ touch file_a
$ touch file_b
$ touch file_c
```

(continues on next page)

(continued from previous page)

```
$ ls
file_a  file_b  file_c  folder1  folder2  folder3
```

These files we have created are all empty. Removing a file is done with the `rm` (remove) command. Please note that on Linux file systems, there is no "Recycle Bin". Any file or folder removed is gone forever and often un-recoverable:

```
$ touch junkfile
$ rm junkfile
```

Moving files with the `mv` command and copying files with the `cp` command works similarly to how you would expect on a Windows or Mac machine. The context around the move or copy operation determines what the result will be. For example, we could move and/or copy files into folders:

```
$ mv file_a folder1/
$ mv file_b folder2/
$ cp file_c folder3/
```

Before listing the results with `ls`, try to guess what the result will be.

```
$ ls
file_c folder1  folder2  folder3
$ ls folder1
file_a
$ ls folder2
file_b
$ ls folder3
file_c
```

Two files have been moved into folders, and `file_c` has been copied - so there is still a copy of `file_c` in the home directory. Move and copy commands can also be used to change the name of a file:

```
$ cp file_c file_c_copy
$ mv file_c file_c_new_name
```

By now, you may have found that Linux is very unforgiving with typos. Generous use of the `<Tab>` key to auto-complete file and folder names, as well as the `<UpArrow>` to cycle back through command history, will greatly improve the experience. As a general rule, try not to use spaces or strange characters in files or folder names. Stick to:

```
A-Z      # capital letters
a-z      # lowercase letters
0-9      # digits
-        # hyphen
_        # underscore
.        # period
```

Before we move on, let's clean up once again by removing the files and folders we have created. Do you remember the command for removing non-empty folders?

```
$ rm -r folder1
$ rm -r folder2
$ rm -r folder3
```

How do we remove `file_c_copy` and `file_c_new_name`?

```
$ rm file_c_copy
$ rm file_c_new_name
```

1.4.4 Looking at the Contents of Files

Everything we have seen so far has been with empty files and folders. We will now start looking at some real data. Navigate to your home directory, then issue the following `cp` command to copy a public file on the server to your local space:

```
$ cd ~    # the tilde ~ is also a shortcut referring to your home directory
$ pwd
/home/wallen
$ cp /usr/share/dict/words .
$ ls
words
```

Try to use `<Tab>` to autocomplete the name of the file. Also, please notice the single dot `.` at the end of the copy command, which indicates that you want to cp the file to `.`, this present location (your home directory).

This `words` file is a standard file that can be found on most Linux operating systems. It contains 479,828 words, each word on its own line. To see the contents of a file, use the `cat` command to print it to screen:

```
$ cat words
1080
10-point
10th
11-point
12-point
16-point
18-point
1st
2
20-point
```

This is a long file! Printing everything to screen is much too fast and not very useful. We can use a few other commands to look at the contents of the file with `more` control:

```
$ more words
```

Press the `<Enter>` key to scroll through line-by-line, or the `<Space>` key to scroll through page-by-page. Press `q` to quit the view, or `<Ctrl+c>` to force a quit if things freeze up. A `%` indicator at the bottom of the screen shows your progress through the file. This is still a little bit messy and fills up the screen. The `less` command has the same effect, but is a little bit cleaner:

```
$ less words
```

Scrolling through the data is the same, but now we can also search the data. Press the `/` forward slash key, and type a word that you would like to search for. The screen will jump down to the first match of that word. The `n` key will cycle through other matches, if they exist.

Finally, you can view just the beginning or the end of a file with the `head` and `tail` commands. For example:

```
$ head words
$ tail words
```

The > and >> shortcuts in Linux indicate that you would like to redirect the output of one of the commands above. Instead of printing to screen, the output can be redirected into a file:

```
$ cat words > words_new.txt
$ head words > first_10_lines.txt
```

A single greater than sign > will redirect and **overwrite** any contents in the target file. A double greater than sign >> will redirect and **append** any output to the end of the target file.

One final useful way to look at the contents of files is with the `grep` command. `grep` searches a file for a specific pattern, and returns all lines that match the pattern. For example:

```
$ grep "banana" words
banana
bananaquit
bananas
cassabanana
```

Although it is not always necessary, it is safe to put the search term in quotes.

1.4.5 Network and File Transfers

In order to login or transfer files to a remote Linux file system, you must know the hostname (unique network identifier) and the username. If you are already on a Linux file system, those are easy to determine using the following commands:

```
$ whoami
wallen
$ hostname -f
isp02.tacc.utexas.edu
```

Given that information, a user would remotely login to this Linux machine using ssh in a Terminal:

```
[local]$ ssh wallen@isp02.tacc.utexas.edu
(enter password)
[isp02]$
```

Windows users would typically use the program **PuTTY** (or another SSH client) to perform this operation. Logging out of a remote system is done using the `logout` command, or the shortcut `<Ctrl+d>`:

```
[isp02]$ logout
[local]$
```

Copying files from your local computer to your home folder on ISP would require the `scp` command (Windows users use a client "WinSCP"):

```
[local]$ scp my_file wallen@isp02.tacc.utexas.edu:/home/wallen/
(enter password)
```

In this command, you specify the name of the file you want to transfer (`my_file`), the username (`wallen`), the hostname (`isp02.tacc.utexas.edu`), and the path you want to put the file (`/home/wallen/`). Take careful notice of the separators including spaces, the `@` symbol, and the `:`.

Copy files from ISP to your local computer using the following:

```
[local]$ scp wallen@isp02.tacc.utexas.edu:/home/wallen/my_file ./
(enter password)
```

Instead of files, full directories can be copied using the "recursive" flag (`scp -r ...`). The `rsync` tool is an advanced copy tool that is useful for synching data between two sites. Although we will not go into depth here, example `rsync` usage is as follows:

```
$ rsync -azv local remote
$ rsync -azv remote local
```

This is just the basics of copying files. See example scp usage and example rsync usage for more info.

1.4.6 Text Editing with VIM

VIM is a text editor used on Linux file systems.

Open a file (or create a new file if it does not exist):

```
$ vim file_name
```

There are two "modes" in VIM that we will talk about today. They are called "insert mode" and "normal mode". In insert mode, the user is typing text into a file as seen through the terminal (think about typing text into TextEdit or Notepad). In normal mode, the user can perform other functions like save, quit, cut and paste, find and replace, etc. (think about clicking the menu options in TextEdit or Notepad). The two main keys to remember to toggle between the modes are `i` and `Esc`.

Entering VIM insert mode:

```
> i
```

Entering VIM normal mode:

```
> Esc
```

A summary of the most important keys to know for normal mode are:

```
# Navigating the file:

arrow keys        move up, down, left, right
    Ctrl+u        page up
    Ctrl+d        page down

         0        move to beginning of line
         $        move to end of line

        gg        move to beginning of file
         G        move to end of file
        :N        move to line N

# Saving and quitting:

        :q        quit editing the file
        :q!       quit editing the file without saving

        :w        save the file, continue editing
        :wq       save and quit
```

1.4.7 Review of Topics Covered

Part 1: Creating and navigating folders

Command	Effect
`pwd`	print working directory
`ls`	list files and directories
`ls -l`	list files in column format
`mkdir dir_name/`	make a new directory
`cd dir_name/`	navigate into a directory
`rmdir dir_name/`	remove an empty directory
`rm -r dir_name/`	remove a directory and its contents
`.` or `./`	refers to the present location
`..` or `../`	refers to the parent directory

Part 2: Creating and manipulating files

Command	Effect
`touch file_name`	create a new file
`rm file_name`	remove a file
`rm -r dir_name/`	remove a directory and its contents
`mv file_name dir_name/`	move a file into a directory
`mv old_file new_file`	change the name of a file
`mv old_dir/ new_dir/`	change the name of a directory
`cp old_file new_file`	copy a file
`cp -r old_dir/ new_dir/`	copy a directory
`<Tab>`	autocomplete file or folder names
`<UpArrow>`	cycle through command history

Part 3: Looking at the contents of files

Command	Effect
`cat file_name`	print file contents to screen
`cat file_name >> new_file`	redirect output to new file
`more file_name`	scroll through file contents
`less file_name`	scroll through file contents
`head file_name`	output beginning of file
`tail file_name`	output end of a file
`grep pattern file_name`	search for 'pattern' in a file
`~/`	shortcut for home directory
`<Ctrl+c>`	force interrupt
`>`	redirect and overwrite
`>>`	redirect and append

Part 4: Network and file transfers

Command	Effect
`hostname -f`	print hostname
`whoami`	print username
`ssh username@hostname`	remote login
`logout`	logout
`scp local remote`	copy a file from local to remote
`scp remote local`	copy a file from remote to local
`rsync -azv local remote`	sync files between local and remote
`rsync -azv remote local`	sync files between remote and local
`<Ctrl+d>`	logout of host

Part 5: Text editing with VIM

Command	Effect
`vim file.txt`	open "file.txt" and edit with `vim`
`i`	toggle to insert mode
`<Esc>`	toggle to normal mode
`<arrow keys>`	navigate the file
`:q`	quit ending the file
`:q!`	quit editing the file without saving
`:w`	save the file, continue editing
`:wq`	save and quit

1.4.8 Additional Resources

- Practice Linux commands safely in a web-based emulator
- This is a good summary of the important commands you need to know
- Practice VIM in a web browser
- Practice VIM on the command line by typing `vimtutor`

1.5 Python Refresher

To be successful in this class, students should be able to:

- Write and execute Python code on the class server
- Use variables, lists, and dictionaries in Python
- Write conditionals using a variety of comparison operators
- Write useful while and for loops
- Arrange code into clean, well organized functions
- Read input from and write output to a file
- Import and use standard and non-standard Python libraries

Topics covered in this module include:

- Data types and variables (ints, floats, bools, strings, type(), print())
- Arithmetic operations (+, -, *, /, **, %, //)

- Lists and dictionaries (creating, interpreting, appending)
- Conditionals and control loops (comparison operators, if/elif/else, while, for, break, continue, pass)
- Functions (defining, passing arguments, returning values)
- File handling (open, with, read(), readline(), strip(), write())
- Importing libraries (import, random, names, pip)

1.5.1 Log in to the Class Server

To log in to `isp02.tacc.utexas.edu`, follow the instructions for your operating system or ssh client below.

Mac / Linux

```
Open the application 'Terminal'
ssh username@isp02.tacc.utexas.edu
(enter password)
```

Windows

```
Open the application 'PuTTY'
enter Host Name: isp02.tacc.utexas.edu
(click 'Open')
(enter username)
(enter password)
```

If you can't access the class server yet, a local or web-based Python 3 environment will work for this guide. However, you will need to access the class server for future lectures.

Try this Python 3 environment in a browser.

Note: For the first few sections below, we will be using the Python interpreter in *interactive mode* to try out different things. Later on when we get to more complex code, we will be saving the code in files (scripts) and invoking the interpreter non-interactively.

1.5.2 Data Types and Variables

Start up the interactive Python interpreter:

```
[isp02]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tip: To exit the interpreter, type `quit()`.

The most common data types in Python are similar to other programming languages. For this class, we probably only need to worry about **integers**, **floats**, **booleans**, and **strings**.

Assign some values to variables by doing the following:

```
>>> my_int = 5
>>> my_float = 5.0
>>> my_bool = True       # or False, notice capital letters
>>> my_string = 'Hello, world!'
```

In Python, you don't have to declare type. Python figures out the type automatically. Check using the `type()` function:

```
>>> type(my_int)
<class 'int'>
>>> type(my_float)
<class 'float'>
>>> type(my_bool)
<class 'bool'>
>>> type(my_string)
<class 'str'>
```

Print the values of each variable using the `print()` function:

```
>> print(my_int)
5
>> print('my_int')
my_int
```

(Try printing the others as well). And, notice what happens when we print with and without single quotes? What is the difference between `my_int` and `'my_int'`?

You can convert between types using a few different functions. For example, when you read in data from a file, numbers are often read as strings. Thus, you may want to convert the string to integer or float as appropriate:

```
>>> str(my_int)       # convert int to string
>>> str(my_float)     # convert float to string
>>> int(my_string)    # convert string to int
>>> float(my_string)  # convert string to float
>>>
>>> value = 5
>>> print(value)
5
>>> type(value)
<class 'int'>
>>> new_value = str(value)
>>> print(new_value)
'5'
>>> type(new_value)
<class 'str'>
```

1.5.3 Arithmetic Operations

Next, we will look at some basic arithmetic. You are probably familiar with the standard operations from other languages:

```
Operator   Function          Example   Result
+          Addition          1+1       2
-          Subtraction       9-5       4
*          Multiplication    2*2       4
```

(continues on next page)

(continued from previous page)

```
/           Division            8/4         2
**          Exponentiation      3**2        9
%           Modulus             5%2         1
//          Floor division      5//2        2
```

Try a few things to see how they work:

```
>>> print(2+2)
>>> print(355/113)
>>> print(10%9)
>>> print(3+5*2)
>>> print('hello' + 'world')
>>> print('some' + 1)
>>> print('number' * 5)
```

Also, carefully consider how arithmetic options may affect type:

```
>>> number1 = 5.0/2
>>> type(number1)
<class 'float'>
>>> print(number1)
2.5
>>> number2 = 5/2
>>> type(number2)
<class 'float'>
>>> print(number2)
2.5
>>> print(int(number2))
2
```

1.5.4 Lists and Dictionaries

Lists are a data structure in Python that can contain multiple elements. They are ordered, they can contain duplicate values, and they can be modified. Declare a list with square brackets as follows:

```
>>> my_shape_list = ['circle', 'triangle', 'square', 'diamond']
>>> type(my_shape_list)
<class 'list'>
>>> print(my_shape_list)
['circle', 'triangle', 'square', 'diamond']
```

Access individual list elements:

```
>>> print(my_shape_list[0])
circle
>>> type(my_shape_list[0])
<class 'str'>
>>> print(my_shape_list[2])
square
```

Create an empty list and add things to it:

```
>>> my_number_list = []
>>> my_number_list.append(5)      # 'append()' is a method of the list class
>>> my_number_list.append(6)
```

(continues on next page)

(continued from previous page)

```
>>> my_number_list.append(2)
>>> my_number_list.append(2**2)
>>> print(my_number_list)
[5, 6, 2, 4]
>>> type(my_number_list)
<class 'list'>
>>> type(my_number_list[1])
<class 'int'>
```

Lists are not restricted to containing one data type. Combine the lists together to demonstrate:

```
>>> my_big_list = my_shape_list + my_number_list
>>> print(my_big_list)
['circle', 'triangle', 'square', 'diamond', 5, 6, 2, 4]
```

Another way to access the contents of lists is by slicing. Slicing supports a start index, stop index, and step taking the form: `mylist[start:stop:step]`. Only the first colon is required. If you omit the start, stop, or :step, it is assumed you mean the beginning, end, and a step of 1, respectively. Here are some examples of slicing:

```
>>> mylist = ['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[0:2])      # returns the first two things
['thing1', 'thing2']
>>> print(mylist[:2])       # if you omit the start index, it assumes the beginning
['thing1', 'thing2']
>>> print(mylist[-2:])      # returns the last two things (omit the stop index and it␣
↪assumes the end)
['thing4', 'thing5']
>>> print(mylist[:])        # returns the entire list
['thing1', 'thing2', 'thing3', 'thing4', 'thing5']
>>> print(mylist[::2])      # return every other thing (step = 2)
['thing1', 'thing3', 'thing5']
```

Note: If you slice from a list, it returns an object of type list. If you access a list element by its index, it returns an object of whatever type that element is. The choice of whether to slice from a list, or iterate over a list by index, will depend on what you want to do with the data.

Dictionaries are another data structure in Python that contain key:value pairs. They are always unordered, they cannot contain duplicate keys, and they can be modified. Create a new dictionary using curly brackets:

```
>>> my_shape_dict = {
...   'most_favorite': 'square',
...   'least_favorite': 'circle',
...   'pointiest': 'triangle',
...   'roundest': 'circle'
... }
>>> type(my_shape_dict)
<class 'dict'>
>>> print(my_shape_dict)
{'most_favorite': 'square', 'least_favorite': 'circle', 'pointiest': 'triangle',
↪'roundest': 'circle'}
>>> print(my_shape_dict['most_favorite'])
square
```

As your preferences change over time, so to can values stored in dictionaries:

```
>>> my_shape_dict['most_favorite'] = 'rectangle'
>>> print(my_shape_dict['most_favorite'])
rectangle
```

Add new key:value pairs to the dictionary as follows:

```
>>> my_shape_dict['funniest'] = 'squircle'
>>> print(my_shape_dict['funniest'])
squircle
```

Many other methods exist to access, manipulate, interpolate, copy, etc., lists and dictionaries. We will learn more about them out as we encounter them later in this course.

1.5.5 Conditionals and Control Loops

Python **comparison operators** allow you to add conditions into your code in the form of `if` / `elif` / `else` statements. Valid comparison operators include:

```
Operator   Comparison                Example   Result
==         Equal                     1==2       False
!=         Not equal                 1!=2       True
>          Greater than              1>2        False
<          Less than                 1<2        True
>=         Greater than or equal to  1>=2       False
<=         Less Than or equal to     1<=2       True
```

A valid conditional statement might look like:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 > num2):                    # notice the colon
...     print('num1 is larger')          # notice the indent
... elif (num2 > num1):
...     print('num2 is larger')
... else:
...     print('num1 and num2 are equal')
```

In addition, conditional statements can be combined with **logical operators**. Valid logical operators include:

```
Operator   Description                          Example
and        Returns True if both are True        a < b and c < d
or         Returns True if at least one is True a < b or c < d
not        Negate the result                    not( a < b )
```

For example, consider the following code:

```
>>> num1 = 10
>>> num2 = 20
>>>
>>> if (num1 < 100 and num2 < 100):
...     print('both are less than 100')
... else:
...     print('at least one of them is not less than 100')
```

While loops also execute according to conditionals. They will continue to execute as long as a condition is True. For example:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )        # literal string interpolation
...     i = i + 1
```

The `break` statement can also be used to escape loops:

```
>>> i = 0
>>>
>>> while (i < 10):
...     print( f'i = {i}' )
...     i = i + 1
...     if (i==5):
...         break
...     else:
...         continue
```

For loops in Python are useful when you need to execute the same set of instructions over and over again. They are especially great for iterating over lists:

```
>>> my_shape_list = ['circle', 'triangle', 'square', 'diamond']
>>>
>>> for shape in my_shape_list:
...     print(shape)
>>>
>>> for shape in my_shape_list:
...     if (shape == 'circle'):
...         pass                    # do nothing
...     else:
...         print(shape)
```

You can also use the `range()` function to iterate over a range of numbers:

```
>>> for x in range(10):
...     print(x)
>>>
>>> for x in range(10, 100, 5):
...     print(x)
>>>
>>> for a in range(3):
...     for b in range(3):
...         for c in range(3):
...             print( f'{a} + {b} + {c} = {a+b+c}' )
```

Note: The code is getting a little bit more complicated now. It will be better to stop running in the interpreter's interactive mode, and start writing our code in Python scripts.

1.5.6 Functions

Functions are blocks of codes that are run only when we call them. We can pass data into functions, and have functions return data to us. Functions are absolutely essential to keeping code clean and organized.

On the command line, use a text editor to start writing a Python script:

```
[isp02]$ vim function_test.py
```

Enter the following text into the script:

```
def hello_world():
    print('Hello, world!')

hello_world()
```

After saving and quiting the file, execute the script (Python code is not compiled - just run the raw script with the `python3` executable):

```
[isp02]$ python3 function_test.py
Hello, world!
```

Note: Future examples from this point on will assume familiarity with using the text editor and executing the script. We will just be showing the contents of the script and console output.

More advanced functions can take parameters and return results:

```
def add5(value):
    return(value + 5)

final_number = add5(10)
print(final_number)
```

```
15
```

Pass multiple parameters to a function:

```
def add5_after_multiplying(value1, value2):
    return( (value1 * value2) + 5)

final_number = add5_after_multiplying(10, 2)
print(final_number)
```

```
25
```

It is a good idea to put your list operations into a function in case you plan to iterate over multiple lists:

```
def print_ts(mylist):
    for x in mylist:
        if (x[0] == 't'):      # a string (x) can be interpreted as a list of chars!
            print(x)

list1 = ['circle', 'triangle', 'square', 'diamond']
list2 = ['one', 'two', 'three', 'four']

```

(continues on next page)

(continued from previous page)

```
print_ts(list1)
print_ts(list2)
```

```
triangle
two
three
```

There are many more ways to call functions, including handing an arbitrary number of arguments, passing keyword / unordered arguments, assigning default values to arguments, and more.

1.5.7 File Handling

The `open()` function does all of the file handling in Python. It takes two arguments - the *filename* and the *mode*. The possible modes are read (`r`), write (`w`), append (`a`), or create (`x`).

For example, to read a file do the following:

```
with open('/usr/share/dict/words', 'r') as f:
    for x in range(5):
        print(f.readline())
```

```
1080

10-point

10th

11-point

12-point
```

Tip: By opening the file with the `with` statement above, you get built in exception handling, and it automatically will close the file handle for you. It is generally recommended as the best practice for file handling.

You may have noticed in the above that there seems to be an extra space between each word. What is actually happening is that the file being read has newline characters on the end of each line (`\n`). When read into the Python script, the original new line is being printed, followed by another newline added by the `print()` function. Stripping the newline character from the original string is the easiest way to solve this problem:

```
with open('/usr/share/dict/words', 'r') as f:
    for x in range(5):
        print(f.readline().strip('\n'))
```

```
1080
10-point
10th
11-point
12-point
```

Read the whole file and store it as a list:

```
words = []

with open('/usr/share/dict/words', 'r') as f:
    words = f.read().splitlines()                # careful of memory usage

for x in range(5):
    print(words[x])
```

```
1080
10-point
10th
11-point
12-point
```

Write output to a new file on the file system; make sure you are attempting to write somwhere where you have permissions to write:

```
my_shapes = ['circle', 'triangle', 'square', 'diamond']

with open('my_shapes.txt', 'w') as f:
    for shape in my_shapes:
        f.write(shape)
```

```
(in my_shapes.txt)
circletrianglesquarediamond
```

You may notice the output file is lacking in newlines this time. Try adding newline characters to your output:

```
my_shapes = ['circle', 'triangle', 'square', 'diamond']

with open('my_shapes.txt', 'w') as f:
    for shape in my_shapes:
        f.write( f'{shape}\n' )
```

```
(in my_shapes.txt)
circle
triangle
square
diamond
```

Now notice that the original line in the output file is gone - it has been overwritten. Be careful if you are using write (`w`) vs. append (`a`).

1.5.8 Importing Libraries

The Python built-in functions, some of which we have seen above, are useful but limited. Part of what makes Python so powerful is the huge number and variety of libraries that can be *imported*. For example, if you want to work with random numbers, you have to import the 'random' library into your code, which has a method for generating random numbers called 'random'.

```
import random

for i in range(5):
    print(random.random())
```

```
0.47115888799541383
0.5202615354150987
0.8892412583071456
0.7467080997595558
0.025668541754695906
```

More information about using the `random` library can be found in the Python docs

Some libraries that you might want to use are not included in the official Python distribution - called the *Python Standard Library*. Libraries written by the user community can often be found on PyPI.org and downloaded to your local environment using a tool called `pip3`.

For example, if you wanted to download the names library and use it in your Python code, you would do the following:

```
[isp02]$ pip3 install --user names
Collecting names
  Downloading https://files.pythonhosted.org/packages/44/4e/
↪f9cb7ef2df0250f4ba3334fbdabaa94f9c88097089763d8e85ada8092f84/names-0.3.0.tar.gz␣
↪(789kB)
    100% || 798kB 1.1MB/s
Installing collected packages: names
  Running setup.py install for names ... done
Successfully installed names-0.3.0
```

Notice the library is installed above with the `--user` flag. The class server is a shared system and non-privileged users can not download or install packages in root locations. The `--user` flag instructs `pip3` to install the library in your own home directory.

```
1 import names
2
3 for i in range(5):
4     print(names.get_full_name())
```

```
Johnny Campbell
Lawrence Webb
Johnathan Holmes
Mary Wang
Jonathan Henry
```

1.5.9 Exercises

Test your understanding of the materials above by attempting the following exercises.

- Create a list of ~10 different integers. Write a function (using modulus and conditionals) to determine if each integer is even or odd. Print to screen each digit followed by the word 'even' or 'odd' as appropriate.
- Using nested for loops and if statements, write a program that iterates over every integer from 3 to 100 (inclusive) and prints out the number only if it is a prime number.
- Create three lists containing 10 integers each. The first list should contain all the integers sequentially from 1 to 10 (inclusive). The second list should contain the squares of each element in the first list. The third list should contain the cubes of each element in the first list. Print all three lists side-by-side in three columns. E.g. the first row should contain 1, 1, 1 and the second row should contain 2, 4, 8.
- Write a script to read in /usr/share/dict/words and print just the last 10 lines of the file. Write another script to only print words beginning with the letters "pyt".

1.5.10 Additional Resources

- The Python Standard Library
- PEP 8 Python Style Guide
- Python3 environment in a browser
- Jupyter Notebooks in a browser

CHAPTER

TWO

WEEK 2: JSON, UNIT TESTING

In this second week of class, we will work hands-on with some essential concepts that we will use throughout the course. This includes working with the Python JSON library and writing and executing unit tests with the Python unittest library.

Please make sure you meet the following prerequisites for this week: (1) you can log in to the class server, write and execute code, (2) you have a strong working knowledge of Linux commands, and (3) you have a strong working knowledge of the basic elements of Python.

2.1 Working with JSON

In this hands-on module, we will learn how to work with the JSON data format. JSON (javascript object notation) is a powerful, flexible, and lightweight data format that we see a lot throughout this course, especially when working with web apps and REST APIs.

After going through this module, students should be able to:

- Identify and write valid JSON
- Load JSON into an object in a Python script
- Loop over and work with elements in a JSON object
- Write JSON to file from a Python script

2.1.1 Sample JSON

Analogous to Python dictionaries, JSON is typically composed of key:value pairs. The universality of this data structure makes it ideal for exchanging information between programs written in different languages and web apps. A simple, valid JSON object may look like this:

```
{
  "key1": "value1",
  "key2": "value2"
}
```

Although less common in this course, simple arrays of information (analogous to Python lists) are also valid JSON:

```
[
  "thing1", "thing2", "thing3"
]
```

JSON offers a lot of flexibility on the placement of white space and newline characters. Types can also be mixed together, forming complex data structures:

```
{
  "department": "COE",
  "number": 332,
  "name": "Software Engineering and Design",
  "inperson": false,
  "instructors": ["Joe", "Charlie", "Brandi", "Joe"],
  "prerequisites": [
    {"course": "COE 322", "instructor": "Victor"},
    {"course": "SDS 322", "instructor": "Victor"}
  ]
}
```

On the class server, navigate to your home directory and make a new folder for this class. Within that folder, make a subfolder for today's module:

```
[local]$ ssh username@isp02.tacc.utexas.edu
(enter password)
[isp02]$ mkdir coe-332/
[isp02]$ cd coe-332/
[isp02]$ mkdir week02-json && cd week02-json
```

Download this sample JSON files into that folder using the `wget` command, or click this link and cut and paste the contents into a file called `states.json`:

```
[isp02]$ wget https://raw.githubusercontent.com/TACC/coe-332-sp21/main/docs/week02/
↪sample-json/states.json
```

Plug this file (or some of the above samples) into an online JSON validator (e.g. JSONLint). Try making manual changes to some of the entries to see what breaks the JSON format.

2.1.2 Load JSON into a Python Script

The `json` Python library is part of the Python Standard Library, meaning it can be imported without having to be installed by pip. Start editing a new Python script using your method of choice:

```
[isp02]$ vim json_ex.py
```

Warning: Do not name your Python script "json.py". If you `import json` when there is a script called "json.py" in the same folder, it will import that instead of the actual `json` library.

The code you need to read in the JSON file of state names and abbreviations into a Python object is:

```
1 import json
2
3 with open('states.json', 'r') as f:
4     states = json.load(f)
```

Only three simple lines! We `import json` from the standard library so that we can work with the `json` class. We use the safe `with open...` statement to open the file we downloaded read-only into a filehandle called `f`. Finally, we use the `load()` method of the `json` class to load the contents of the JSON file into our new `states` object.

EXERCISE

Try out some of these calls to the `type()` function on the new `states` object that you loaded. Also `print()` each of these as necessary to be sure you know what each is. Be able to explain the output of each call to `type()` and `print()`.

```
import json

with open('states.json', 'r') as f:
    states = json.load(f)

type(states)
type(states['states'])
type(states['states'][0])
type(states['states'][0]['name'])
type(states['states'][0]['name'][0])

print(states)
print(states['states'])
print(states['states'][0])
print(states['states'][0]['name'])
print(states['states'][0]['name'][0])
```

Tip: Consider doing this in the Python interpreter's interactive mode instead of in a script.

2.1.3 Working with JSON

As we have seen, the JSON object we loaded contains state names and abbreviations. In the US, official state abbreviations are unique, two-letter identifiers. Let's write a few functions to help us validate whether our state abbreviations follow the rules or not.

First, write a function to check whether there are exactly two characters in each of the abbreviations. Call that function, and have it return a message about whether the abbreviation passes or fails the test.

```
import json

def check_char_count(mystr):
    if ( len(mystr) == 2 ):
        return( f'{mystr} count passes' )
    else:
        return( f'{mystr} count FAILS' )

with open('states.json', 'r') as f:
    states = json.load(f)

for i in range(50):
    print(check_char_count( states['states'][i]['abbreviation']))
```

Next, write a function to check whether both characters are actually uppercase letters, and not something else like a number or a special character or a lowercase letter. Again, have it return a pass or fail message as appropriate.

```
import json

def check_char_count(mystr):
```

(continues on next page)

(continued from previous page)

```
    if (len(mystr) == 2):
        return( f'{mystr} count passes' )
    else:
        return( f'{mystr} count FAILS' )

def check_char_type(mystr):
    if (mystr.isalpha() and mystr.isupper()):
        return( f'{mystr} type passes' )
    else:
        return( f'{mystr} type FAILS' )

with open('states.json', 'r') as f:
    states = json.load(f)

for i in range(50):
    print(check_char_count( states['states'][i]['abbreviation']))
    print(check_char_type( states['states'][i]['abbreviation']))
```

EXERCISE

Write a third function to check that the first character of each abbreviation matches the first character of the corresponding state. Return pass or fail messages as appropriate.

2.1.4 Write JSON to File

Finally, in a new script, we will create an object that we can write to a new JSON file.

```
import json

data = {}
data['class'] = 'COE332'
data['title'] = 'Software Engineering and Design'
data['subjects'] = []
data['subjects'].append( {'week': 1, 'topic': ['linux', 'python']} )
data['subjects'].append( {'week': 2, 'topic': ['json', 'unittest', 'git']} )

with open('class.json', 'w') as out:
    json.dump(data, out, indent=2)
```

Notice that most of the code in the script above was simply assembling a normal Python dictionary. The `json.dump()` method only requires two arguments - the object that should be written to file, and the filehandle. The `indent=2` argument is optional, but it makes the output file looks a little nicer and easier to read.

Inspect the output file and paste the contents into an online JSON validator.

2.1.5 Additional Resources

- Reference for the JSON library
- Validate JSON with JSONLint

2.2 Unit Tests in Python

In the previous module, we wrote a simple script with three functions used to validate the state abbreviations in a JSON file. In this module, we will use the Python `unittest` library to write unit tests for those functions.

After working through this module, students should be able to:

- Find the documentation for the Python `unittest` library
- Identify parts of code that should be tested and appropriate assert methods
- Write and run reasonable unit tests

2.2.1 Getting Started

Unit tests are designed to test small components (e.g. individual functions) of your code. They should demonstrate that things that are expected to work actually do work, and things that are expected to break raise appropriate errors. The Python `unittest` unit testing framework supports test automation, set up and shut down code for tests, and aggregation of tests into collections. It is built in to the Python Standard Library and can be imported directly. Find the documentation here. The best way to see how it works is to see it applied to a real example.

Pull a copy of the Python script from the previous section if you don't have one already.

We need to make a small organizational change to this code in order to make it work with the test suite (this is good organizational practice anyway). We will move everything that is not already a function into a new `main()` function:

```
import json

def check_char_count(mystr):
    if ( len(mystr) == 2 ):
        return( f'{mystr} count passes' )
    else:
        return( f'{mystr} count FAILS' )

def check_char_type(mystr):
    if ( mystr.isupper() and mystr.isalpha() ):
        return( f'{mystr} type passes' )
    else:
        return( f'{mystr} type FAILS' )

def check_char_match(str1, str2):
    if ( str1[0] == str2[0] ):
        return( f'{str1} match passes' )
    else:
        return( f'{str1} match FAILS' )

def main():
    with open('states.json', 'r') as f:
        states = json.load(f)

```

(continues on next page)

(continued from previous page)

```
    for i in range(50):
        print(check_char_count( states['states'][i]['abbreviation'] ))
        print(check_char_type( states['states'][i]['abbreviation'] ))
        print(check_char_match( states['states'][i]['abbreviation'], states['states
↪'][i]['name'] ))

if __name__ == '__main__':
    main()
```

The last two lines make it so the `main()` function is only called if this script is executed directly, and not if it is imported into another script.

2.2.2 Break a Function

The function in this example Python script are simple, but can be easily broken if not used as intended. Use the Python interactive interpreter to import the functions we wrote and find out what breaks them:

```
>>> from json_ex import check_char_count        # that was easy!
>>>
>>> check_char_count('AA')     # this is supposed to pass
'AA count passes'
>>> check_char_count('AAA')    # this is supposed to fail
'AAA count FAILS'
>>> check_char_count(12)       # what if we send an int instead of a string?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/wallen/test2/json_ex.py", line 4, in check_char_count
    if ( len(mystr) == 2 ):
TypeError: object of type 'int' has no len()
>>> check_char_count([12, 34])                  # ... uh oh
'[12, 34] count passes'
```

Everything looked good until we sent our function a **list with two elements**. The function we wrote just checks the length of whatever we sent as an argument, but we never intended lists to pass. So now we need to do two things:

- Write up the above tests in an automated way
- Fix our function so lists don't pass through

Create a new file called `test_json_ex.py` and start writing code to automate the above tests:

Tip: It is common Python convention to name a test file the same name as the script you are testing, but with the `test_` prefix added at the beginning.

```
import unittest
from json_ex import check_char_count

class TestJsonEx(unittest.TestCase):

    def test_check_char_count(self):
        self.assertEqual(check_char_count('AA'), 'AA count passes')
        self.assertEqual(check_char_count('AAA'), 'AAA count FAILS')

if __name__ == '__main__':
    unittest.main()
```

In the simplest case above, we do several things:

- Import the `unittest` framework
- Import the function (`check_char_count`) we want to test
- Create a class for testing our application (json_ex) and subclass `unittest.TestCase`
- Define a method for testing a specific function (`check_char_count`) from our application
- Write tests to check that certain calls to our function return what we expect
- Wrap the `unittest.main()` function at the bottom so we can call this script

The key part of the above test are the `assertEqual` methods. The test will only pass if the two parameters passed to that method are equal. Execute the script to run the tests:

```
[isp02]$ python3 test_json_ex.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Success! Next, we can start to look at edge cases. If you recall above, sending an `int` to this function raised a `TypeError`. This is good and expected behavior! We can use the `assertRaises` method to make sure that happens:

```
def test_check_char_count(self):
    self.assertEqual(check_char_count('AA'), 'AA count passes')
    self.assertEqual(check_char_count('AAA'), 'AAA count FAILS')
    self.assertRaises(TypeError, check_char_count, 1)
    self.assertRaises(TypeError, check_char_count, True)
    self.assertRaises(TypeError, check_char_count, ['AA', 'BB'])
```

Tip: How do we know what parameters to pass to the `assertRaises` method? Check the documentation of course!

Run it again to see what happens:

```
[isp02]$ python3 test_json_ex.py
F
======================================================================
FAIL: test_check_char_count (__main__.TestJsonEx)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_json_ex.py", line 11, in test_check_char_count
    self.assertRaises(TypeError, check_char_count, ['AA', 'BB'])
AssertionError: TypeError not raised by check_char_count

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Our test failed because we are trying to assert that sending our function a list should result in a TypeError. But, that's not what happened - in fact sending our function a list resulted in a pass without error.

2.2.3 Fix a Function

We need to modify our function in `json_ex.py` to handle edge cases better. We don't want to pass anything sent to this function other than a two-character **string**. So, let's modify our function and add an assert statement to make sure the thing passed to the function is in fact a string:

```
def check_char_count(mystr):

    assert isinstance(mystr, str), 'Input to this function should be a string'

    if ( len(mystr) == 2 ):
        return( f'{mystr} count passes' )
    else:
        return( f'{mystr} count FAILS' )
```

Assert statements are a convenient way to put checks in code with helpful print statements for debugging. Run `json_ex.py` again to make sure it is still working, then run the test suite again:

```
[isp02]$ python3 test_json_ex.py
F
======================================================================
FAIL: test_check_char_count (__main__.TestJsonEx)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_json_ex.py", line 9, in test_check_char_count
    self.assertRaises(TypeError, check_char_count, 1)
AssertionError: Input to this function should be a string

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Whoops! The test is still failing. This is because we are trying to enforce a `TypeError` when we send our function an `int`. However, with the new assert statement in our function, we are raising an `AssertionError` before the `TypeError` ever has a chance to crop up. We must modify our tests to now look for `AssertionErrors`.

```
def test_check_char_count(self):
    self.assertEqual(check_char_count('AA'), 'AA count passes')
    self.assertEqual(check_char_count('AAA'), 'AAA count FAILS')
    self.assertRaises(AssertionError, check_char_count, 1)
    self.assertRaises(AssertionError, check_char_count, True)
    self.assertRaises(AssertionError, check_char_count, ['AA', 'BB'])
```

Then run the test suite one more time:

```
[isp02]$ python3 test_json_ex.py
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Success! The test for our first function is passing. Our test suite essentially documents our intent for the behavior of the `check_char_count()` function. And, if ever we change the code in that function, we can see if the behavior we intend still passes the test.

2.2.4 Another Function, Another Test

The next function in our original code is `check_char_type()`, which checks to see that the passed string consists of uppercase letters only. This function is already pretty fail safe because it is using built-in string methods (`isupper()` and `isalpha()`) to do the checking. These already have internal error handling, so we can probably get away with a few simple tests and no changes to our original function.

Add the following lines to the `test_json_ex.py`:

```
import unittest
from json_ex import check_char_count
from json_ex import check_char_type

class TestJsonEx(unittest.TestCase):

    def test_check_char_count(self):
        self.assertEqual(check_char_count('AA'), 'AA count passes')
        self.assertEqual(check_char_count('AAA'), 'AAA count FAILS')
        self.assertRaises(AssertionError, check_char_count, 1)
        self.assertRaises(AssertionError, check_char_count, True)
        self.assertRaises(AssertionError, check_char_count, ['AA', 'BB'])

     def test_check_char_type(self):
         self.assertEqual(check_char_type('AA'), 'AA type passes')
         self.assertEqual(check_char_type('Aa'), 'Aa type FAILS')
         self.assertEqual(check_char_type('aa'), 'aa type FAILS')
         self.assertEqual(check_char_type('A1'), 'A1 type FAILS')
         self.assertEqual(check_char_type('a1'), 'a1 type FAILS')
         self.assertRaises(AttributeError, check_char_type, 1)
         self.assertRaises(AttributeError, check_char_type, True)
         self.assertRaises(AttributeError, check_char_type, ['AA', 'BB'])

if __name__ == '__main__':
    unittest.main()
```

The `isupper()` and `isalpha()` methods only work on strings - if you try them on anything else, they will automatically return an `AttributeError`. We can confirm this with our tests.

Run the tests again to be sure you have two passing tests:

```
[isp02]$ python3 test_json_ex.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

EXERCISE

Focusing on the `assertEqual()` and `assertRaises()` methods, write reasonable tests for the final function - `check_char_match()`.

2.2.5 Additional Resources

- Python unittest documentation
- Exceptions in Python

CHAPTER

THREE

WEEK 3: VERSION CONTROL, INTRO TO CONTAINERS

In this third week of class, we will familiarize ourselves with version control by working with Git and GitHub. We will also see our first introduction to containers through Docker and Docker Hub.

If you don't have them already, please create accounts on GitHub and Docker Hub, both of which we will use extensively in this class. (You can use existing personal accounts, or if you prefer you can make new accounts specifically for this class).

3.1 Version Control with Git

In this module, we will look at the version control system **Git**. Of the numerous version control systems available (Git, Subversion, CVS, Mercurial), Git seems to be the most popular, and we generally find that it is great for:

- Collaborating with others on code
- Supporting multiple concurrent versions (branches)
- Tagging releases or snapshots in time
- Restoring previous versions of files
- What it lacks in user-friendliness it makes up for in good documentation
- Intuitive web platforms available

After going through this module, students should be able to:

- Create a new Git repository hosted on GitHub
- Clone a repository, commit and push changes to the repository
- Track the history of changes in files in a Git repository
- Work collaboratively with others on the content in a Git repository
- Demonstrate a basic understanding of forking, branching, and tags

GitHub is a web platform where you can host and share Git repositories ("repos"). Repositories can be public or private. Much of what we will do with this section requires you to have a GitHub account.

3.1.1 The Basics of Git

Version control systems start with a base version of the document and then record changes you make each step of the way. You can think of it as a recording of your progress: you can rewind to start at the base document and play back each change you made, eventually arriving at your more recent version.

Fig. 1: Changes are saved sequentially.

Once you think of changes as separate from the document itself, you can then think about "playing back" different sets of changes on the base document, ultimately resulting in different versions of that document. For example, two users can make independent sets of changes on the same document.

Fig. 2: Different versions can be saved.

Unless there are conflicts, you can even incorporate two sets of changes into the same base document.

Fig. 3: Multiple versions can be merged.

A version control system is a tool that keeps track of these changes for us, effectively creating different versions of our files. It allows us to decide which changes will be made to the next version (each record of these changes is called a "commit", and keeps useful metadata about them. The complete history of commits for a particular project and their metadata make up a "repository". Repositories can be kept in sync across different computers, facilitating collaboration among different people.

3.1.2 Setting up Git

Log on to the class ISP server and check which version of Git is in your `PATH`.

```
[local]$ ssh username@isp02.tacc.utexas.edu   # use your account
(enter password)

[isp02]$ which git
/opt/apps/git/2.24.1/bin/git
$ git --version
git version 1.8.3.1
```

When we use Git on a new computer for the first time, we need to configure a few things. Below are a few examples of configurations we will set as we get started with Git:

- Our name and email address,
- And that we want to use these settings globally (i.e. for every project).

On a command line, Git commands are written as `git verb`, where `verb` is what we actually want to do. Here is how we set up our environment:

```
[isp02]$ git config --global user.name "Joe Allen"
[isp02]$ git config --global user.email "wallen@tacc.utexas.edu"
```

Please use your own name and email address. This user name and email will be associated with your subsequent Git activity, which means that any changes pushed to GitHub, Bitbucket, GitLab or another Git host server in the future will include this information.

Tip: A key benefit of Git is that it is platform agnostic. You can use it equally to interact with the same files from your laptop, from a lab computer, or from a cluster.

3.1.3 Create a New Repository on the Command Line

First, let's navigate back to our folder from the JSON module:

```
[isp02]$ cd ~/coe-332/week02-json
```

Then we will use a Git command to initialize this directory as a new Git repository - or a place where Git can start to organize versions of our files.

```
[isp02]$ git init
Initialized empty Git repository in /home/wallen/coe-332/week02-json/.git/
```

If we use `ls -a`, we can see that Git has created a hidden directory called `.git`:

```
[isp02]$ ls -a
./  ../  class.json  .git/  json_ex.py  json_write.py  states.json
```

Use the `find` command to get a overview of the contents of the `.git/` directory:

```
[isp02]$ find .git/
.git
.git/refs
.git/refs/heads
.git/refs/tags
.git/branches
.git/description
.git/hooks
.git/hooks/applypatch-msg.sample
.git/hooks/commit-msg.sample
.git/hooks/post-update.sample
.git/hooks/pre-applypatch.sample
.git/hooks/pre-commit.sample
.git/hooks/pre-push.sample
.git/hooks/pre-rebase.sample
.git/hooks/prepare-commit-msg.sample
.git/hooks/update.sample
.git/info
.git/info/exclude
.git/HEAD
.git/config
.git/objects
.git/objects/pack
.git/objects/info
```

Git uses this special sub-directory to store all the information about the project, including all files and sub-directories located within the project's directory. If we ever delete the `.git` sub-directory, we will lose the project's history. We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
[isp02]$ git status
# On branch main
#
```

(continues on next page)

(continued from previous page)

```
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       class.json
#       json_ex.py
#       json_write.py
#       states.json
nothing added to commit but untracked files present (use "git add" to track)
```

Note: If you are using a different version of `git`, the exact wording of the output might be slightly different.

EXERCISE

- Explore the files and folders in the `.git/` directory
- Can you find a file with your name and e-mail in it? How did it get there?

3.1.4 Tracking Changes

We will use this repository track some changes we are about to make to our example JSON parsing scripts. Above, Git mentioned that it found several "Untracked files". This means there are files in this current directory that Git isn't keeping track of. We can instruct Git to start tracking a file using `git add`:

```
[isp02]$ git add json_ex.py
[isp02]$ git status
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   json_ex.py
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       class.json
#       json_write.py
#       states.json
```

3.1.5 Commit Changes to the Repo

Git now knows that it's supposed to keep track of `json_ex.py`, but it hasn't recorded these changes as a commit yet. To get it to do that, we need to run one more command:

```
[isp02]$ git commit -m "started tracking json example script"
[main (root-commit) 344ec9f] started tracking json example script
 1 file changed, 29 insertions(+)
 create mode 100644 json_ex.py
```

When we run `git commit`, Git takes everything we have told it to save by using `git add` and stores a copy permanently inside the special `.git` directory. This permanent copy is called a "commit" (or "revision") and its short identifier is `344ec9f`. Your commit may have another identifier.

We use the `-m` flag ("m" for "message") to record a short, descriptive, and specific comment that will help us remember later on what we did and why. Good commit messages start with a brief (<50 characters) statement about the changes made in the commit. Generally, the message should complete the sentence "If applied, this commit will" <*commit message here*>. If you want to go into more detail, add a blank line between the summary line and your additional notes. Use this additional space to explain why you made changes and/or what their impact will be.

If we run `git status` now:

```
[isp02]$ git status
# On branch main
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       class.json
#       json_write.py
#       states.json
nothing added to commit but untracked files present (use "git add" to track)
```

We find three remaining untracked files.

EXERCISE

Do a `git add <file>` followed by a `git commit -m "descriptive message"` for each file, one by one. Also do a `git status` in between each command.

3.1.6 Check the Project History

If we want to know what we've done recently, we can ask Git to show us the project's history using `git log`:

```
[isp02]$ git log
commit 13e07d9dd6a6d3b47f4b7537035c9c532fb7cf4e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:29 2021 -0600

    adding states.json

commit f20159ea98b276ff300b018fa420b514e53e2042
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:15 2021 -0600

    adding json_write.py
```

(continues on next page)

(continued from previous page)

```
commit 3d5d6e2c6d23aa4fb3b800b535db6a228759866e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:03 2021 -0600

    adding class.json

commit 344ec9fde550c6e009697b07298919946ff991f9
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:00:17 2021 -0600

    started tracking json example script
```

The command `git log` lists all commits made to a repository in reverse chronological order. The listing for each commit includes:

- the commit's full identifier (which starts with the same characters as the short identifier printed by the `git commit` command earlier),
- the commit's author,
- when it was created,
- and the log message Git was given when the commit was created.

3.1.7 Making Further Changes

Now suppose we make a change to one of the files we are tracking. Edit the `json_ex.py` script your favorite text editor and add some random comments into the script:

```
[isp02]$ vim json_ex.py
# make some changes in the script
# save and quit
```

When we run `git status` now, it tells us that a file it already knows about has been modified:

```
[isp02]$ git status
# On branch main
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   json_ex.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The last line is the key phrase: "no changes added to commit". We have changed this file, but we haven't told Git we will want to save those changes (which we do with `git add`) nor have we saved them (which we do with `git commit`). So let's do that now. It is good practice to always review our changes before saving them. We do this using `git diff`. This shows us the differences between the current state of the file and the most recently saved version:

```
[isp02]$ git diff json_ex.py
diff --git a/json_ex.py b/json_ex.py
index 5d986e9..21877cb 100644
--- a/json_ex.py
+++ b/json_ex.py
```

(continues on next page)

(continued from previous page)

```
@@ -18,7 +18,7 @@ def check_char_match(str1, str2):
     else:
         return( f'{str1} match FAILS' )

-
+# open the json file and load into dict
 with open('states.json', 'r') as f:
     states = json.load(f)
```

The output is cryptic because it is actually a series of commands for tools like editors and `patch` telling them how to reconstruct one file given the other. If we break it down into pieces:

- The first line tells us that Git is producing output similar to the Unix `diff` command comparing the old and new versions of the file.
- The second line tells exactly which versions of the file Git is comparing: `5d986e9` and `21877cb` are unique computer-generated labels for those versions.
- The third and fourth lines once again show the name of the file being changed.
- The remaining lines are the most interesting, they show us the actual differences and the lines on which they occur. In particular, the + marker in the first column shows where we added lines.

After reviewing our change, it's time to commit it:

```
[isp02]$ git add json_ex.py
[isp02]$ git commit -m "added a descriptive comment"
[main 8d5f563] added a descriptive comment
 1 file changed, 1 insertion(+), 1 deletion(-)
[isp02]$ git status
# On branch main
nothing to commit, working directory clean
```

Git insists that we add files to the set we want to commit before actually committing anything. This allows us to commit our changes in stages and capture changes in logical portions rather than only large batches. For example, suppose we're adding a few citations to relevant research to our thesis. We might want to commit those additions, and the corresponding bibliography entries, but *not* commit some of our work drafting the conclusion (which we haven't finished yet).

3.1.8 Directories in Git

There are a couple important facts you should know about directories in Git. First, Git does not track directories on their own, only files within them. Try it for yourself:

```
[isp02]$ mkdir directory
[isp02]$ git status
[isp02]$ git add directory
[isp02]$ git status
```

Note, our newly created empty directory `directory` does not appear in the list of untracked files even if we explicitly add it (*via* `git add`) to our repository.

Second, if you create a directory in your Git repository and populate it with files, you can add all files in the directory at once by:

```
[isp02]$ git add <directory-with-files>
```

Tip: A trick for tracking an empty directory with Git is to add a hidden file to the directory. People sometimes will label this `.gitcanary`. Adding and committing that file to the repo's history will cause the directory it is in to also be tracked.

3.1.9 Restoring Old Versions of Files

We can save changes to files and see what we've changed — now how can we restore older versions of things? Let's suppose we accidentally overwrite our file:

```
[isp02]$ echo "" > json_ex.py
[isp02]$ cat json_ex.py
```

Now `git status` tells us that the file has been changed, but those changes haven't been staged:

```
[isp02]$ git status
# On branch main
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   json_ex.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We can put things back the way they were by using `git checkout` and referring to the *most recent commit* of the working directory by using the identifier `HEAD`:

```
[isp02]$ git checkout HEAD json_ex.py
[isp02]$ cat json_ex.py
import json
...etc
```

As you might guess from its name, `git checkout` checks out (i.e., restores) an old version of a file. In this case, we're telling Git that we want to recover the version of the file recorded in `HEAD`, which is the last saved commit. If we want to go back even further, we can use a commit identifier instead:

```
[isp02]$ git log
commit 8d5f563fa20060f4fbe2e10ec5cbc3c22fe92559
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:15:46 2021 -0600

 added a descriptive comment

commit 13e07d9dd6a6d3b47f4b7537035c9c532fb7cf4e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:29 2021 -0600

 adding states.json

commit f20159ea98b276ff300b018fa420b514e53e2042
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:15 2021 -0600

 adding json_write.py
```

(continues on next page)

(continued from previous page)

```
commit 3d5d6e2c6d23aa4fb3b800b535db6a228759866e
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:06:03 2021 -0600

 adding class.json

commit 344ec9fde550c6e009697b07298919946ff991f9
Author: Joe Allen <wallen@tacc.utexas.edu>
Date:   Wed Jan 27 23:00:17 2021 -0600

 started tracking json example script
```

```
[isp02]$ git checkout 344ec9f json_ex.py
# now you have a copy of json_ex.py without that comment we added
```

Again, we can put things back the way they were by using `git checkout`:

```
[isp02]$ git checkout HEAD json_ex.py
# back to the most recent version
```

3.1.10 Link a Local Repository to GitHub

Version control really comes into its own when we begin to collaborate with other people. We already have most of the machinery we need to do this; the only thing missing is to copy changes from one repository to another.

Systems like Git allow us to move work between any two repositories. In practice, though, it's easiest to use one copy as a central hub, and to keep it on the web rather than on someone's laptop. Most programmers use hosting services like GitHub, Bitbucket, or GitLab to hold those main copies.

Let's start by sharing the changes we've made to our current project with the world. Log in to GitHub, then click on the icon in the top right corner to create a new repository:

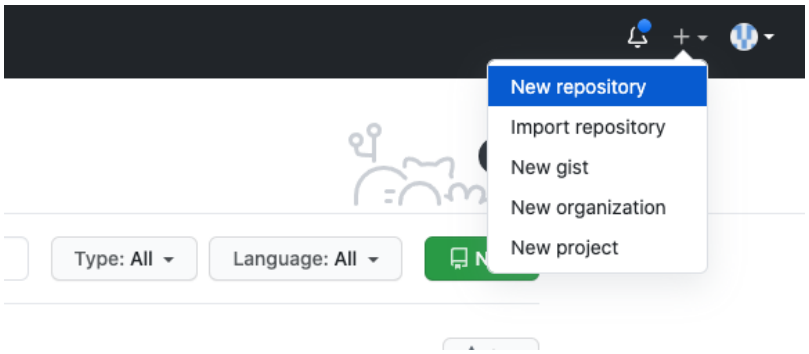


Fig. 4: Click 'New repository'.

As soon as the repository is created, GitHub displays a page with a URL and some information on how to configure your local repository. Provide a name for your new repository like `json-parser` (or whatever you want).

Note that our local repository still contains our earlier work on `json_ex.py` and other files, but the remote repository on GitHub doesn't contain any memory of `json_ex.py` yet. The next step is to connect the two repositories. We do this by making the GitHub repository a "remote" for the local repository. The home page of the repository on GitHub includes the string we need to identify it:

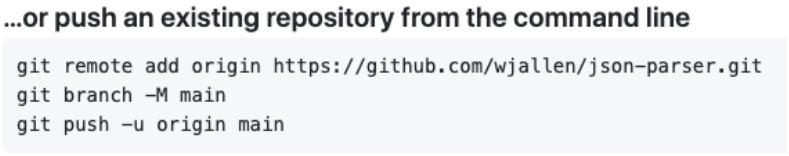


Fig. 5: Follow the instructions for pushing an existing repository.

Back on ISP in the local Git repo, link it to the repo on GitHub and confirm the link was created:

```
[isp02]$ git remote add origin https://github.com/wjallen/json-parser.git
[isp02]$ git remote -v
origin  https://github.com/wjallen/json-parser.git (fetch)
origin  https://github.com/wjallen/json-parser.git (push)
```

Attention: Make sure to use the URL for your repository instead of the one listed here.

The name `origin` is a local nickname for your remote repository. We could use something else if we wanted to, but `origin` is by far the most common choice.

Once the nickname `origin` is set up, this command will push the changes from our local repository to the repository on GitHub:

```
[isp02]$ git branch -M main
[isp02]$ git push -u origin main
Username for 'https://github.com': wjallen
Password for 'https://wjallen@github.com':
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 2.30 KiB | 0 bytes/s, done.
Total 15 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/wjallen/json-parser.git
 * [new branch]      main -> main
Branch main set up to track remote branch main from origin.
```

3.1.11 Clone the Repository

Spend a few minutes browsing the web interface for GitHub. Now, anyone can make a full copy of `my_first_repo` including all the commit history by performing:

```
[isp02]$ git clone https://github.com/wjallen/json-parser
Cloning into 'json-parser'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 15 (delta 4), reused 15 (delta 4), pack-reused 0
Unpacking objects: 100% (15/15), done.
```

3.1.12 Collaborating with Others

A public platform like GitHub makes it easier than ever to collaborate with others on the content of a repository. You can have as many local copies of a repository as you want, but there is only one "origin" repository - the repository hosted on GitHub. Other repositories may fall behind the origin, or have changes that are ahead of the origin. A common model for juggling multiple repositories where separate individuals are working on different features is the GitFlow model:

Some important definitions (most can easily be managed right in the GitHub web interface):

FORK

A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.

BRANCH

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or main branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the main branch to publish your changes. For more information, see About branches.

TAG

Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on).

PULL REQUEST / MERGE REQUEST

Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests ach have their own discussion forum. For more information, see About pull requests.

OTHER CONSIDERATIONS

Most repos will also contain a few standard files in the top directory, including:

README.md: The landing page of your repository on GitHub will display the contents of README.md, if it exists. This is a good place to describe your project and list the appropriate citations.

LICENSE.txt: See if your repository needs a license here.

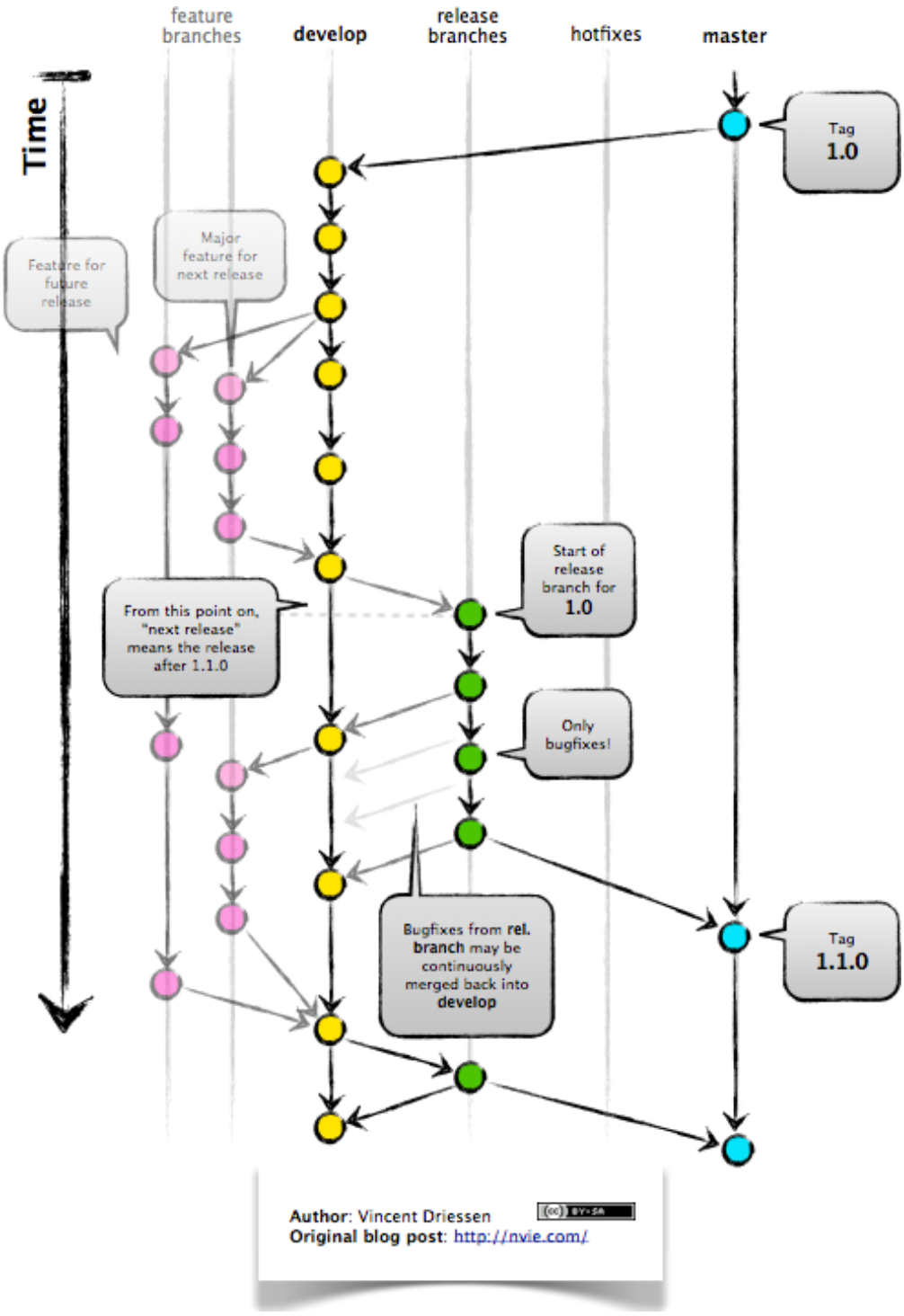


Fig. 6: GitFlow model

3.1.13 Additional Resources

- Some of the materials in this module were based on Software Carpentry DOI: 10.5281/zenodo.57467.
- GitHub Glossary
- About Branches
- About Pull Requests
- About Licenses
- GitFlow Model
- More on different git workflows

3.2 Introduction to Containers

Containers are an important common currency for app development, web services, scientific computing, and more. Containers allow you to package an application along with all of its dependencies, isolate it from other applications and services, and deploy it consistently and reproducibly and *platform-agnostically*. In this introductory module, we will learn about containers and their uses, in particular the containerization platform **Docker**.

After going through this module, students should be able to:

- Describe what a container is
- Use essential docker commands
- Find and pull existing containers from Docker Hub
- Run containers interactively and non-interactively

3.2.1 What is a Container?

- A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package.
- Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space, hence are *lightweight* and have *low overhead*.
- Containers ensure *portability* and *reproducibility* by isolating the application from environment.

3.2.2 How is a Container Different from a VM?

Virtual machines enable application and resource isolation, run on top of a hypervisor (high overhead). Multiple VMs can run on the same physical infrastructure - from a few to dozens depending on resources. VMs take up more disk space and have long start up times (~minutes).

Containers enable application and resource isolation, run on top of the host operating system. Many containers can run on the same physical infrastructure - up to 1,000s depending on resources. Containers take up less disk space than VMs and have very short start up times (~100s of ms).

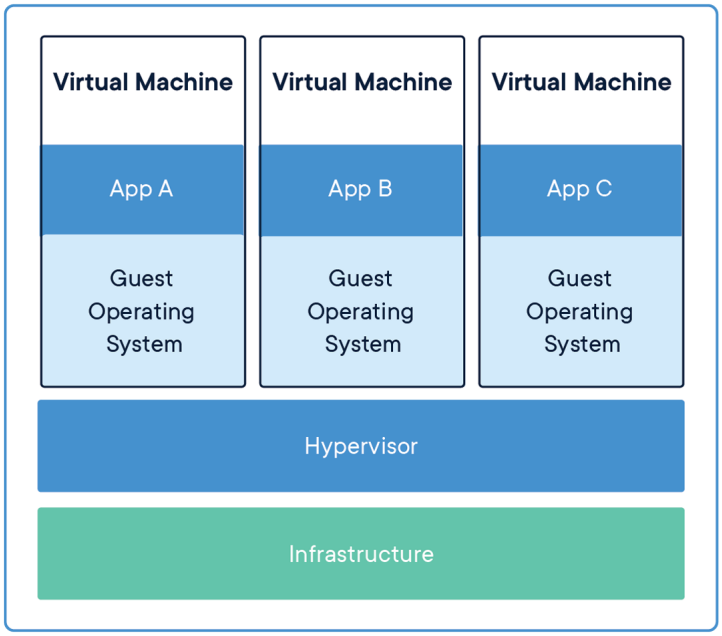


Fig. 7: Applications isolated by VMs.

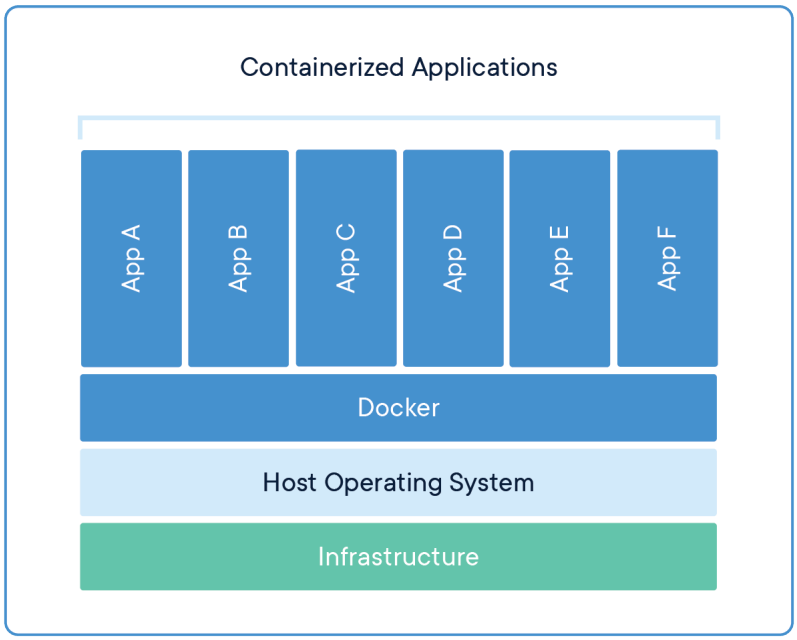


Fig. 8: Applications isolated by containers.

3.2.3 Docker

Docker is a containerization platform that uses OS-level virtualization to package software and dependencies in deliverable units called containers. It is by far the most common containerization platform today, and most other container platforms are compatible with Docker. (E.g. Singularity and Shifter are two containerization platforms you'll find in HPC environments).

We can find existing containers at:

1. Docker Hub
2. Quay.io
3. BioContainers

3.2.4 Some Quick Definitions

CONTAINER

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. Containers includes everything from the operating system, user-added files, metadata.

IMAGE

A Docker image is a read-only file used to produce Docker containers. It is comprised of layers of other images, and any changes made to an image can only be saved and propagated on by adding new layers. The "base image" is the bottom-most layer that does not depend on any other layer and typically defines, e.g., the operating system for the container. Running a Docker image creates an instance of a Docker container.

DOCKERFILE

The Dockerfile is a recipe for creating a Docker image. They are simple, usually short plain text files that contain a sequential set of commands (*a recipe*) for installing and configuring your application and all of its dependencies. The Docker command line interface is used to "build" an image from a Dockerfile.

IMAGE REGISTRY

The Docker images you build can be stored in online image registries, such as Docker Hub. (It is similar to the way we store Git repositories on GitHub.) Image registries support the notion of tags on images to identify specific versions of images. It is mostly public, and many "official" images can be found.

3.2.5 Summing it Up

If you are developing an app or web service, you will almost certainly want to work with containers. First you must either **build** an image from a Dockerfile, or **pull** an image from a public registry. Then, you **run** (or deploy) an instance of your image into a container. The container represents your app or web service, running in the wild, isolated from other apps and services.

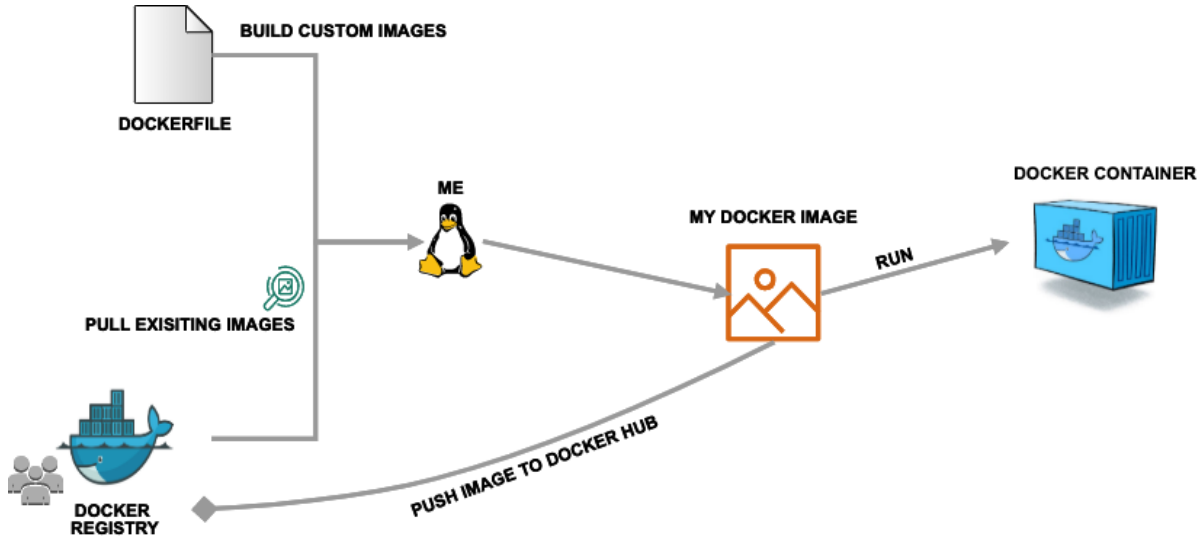


Fig. 9: Simple Docker workflow.

3.2.6 Getting Started With Docker

Much like the `git` command line tools, the `docker` command line tools follow the syntax: `docker <verb> <parameters>`. Discover all the verbs available by typing `docker --help`, and discover help for each verb by typing `docker <verb> --help`. Open up your favorite terminal, log in to the class server, and try running the following:

```
[isp02]$ docker version
Client: Docker Engine - Community
 Version:           20.10.3
 API version:       1.41
 Go version:        go1.13.15
 Git commit:        48d30b5
 Built:             Fri Jan 29 14:34:14 2021
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.3
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       46229ca
  Built:            Fri Jan 29 14:32:37 2021
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.3
  GitCommit:        269548fa27e0089a8b8278fc4fc781d7f65a939b
 runc:
  Version:          1.0.0-rc92
  GitCommit:        ff819c7e9184c13b7c2607fe6c30ae19403a7aff
 docker-init:
  Version:          0.19.0
```

(continues on next page)

(continued from previous page)

```
 GitCommit:        de40ad0
```

Warning: Please let the instructors know if you get any errors on issuing the above command.

EXERCISE

Take a few minutes to run `docker --help` and a few examples of `docker <verb> --help` to make sure you can find and read the help text.

3.2.7 Working with Images from Docker Hub

To introduce ourselves to some of the most essential Docker commands, we will go through the process of listing images that are currently available on the ISP server, we will pull a 'hello-world' image from Docker Hub, then we will run the 'hello-world' image to see what it says.

List images on the ISP server with the `docker images` command. This peeks into the Docker daemon, which is shared by all users on this system, to see which images are available, when they were created, and how large they are:

```
[isp02]$ docker images
REPOSITORY         TAG       IMAGE ID       CREATED          SIZE
final_web          latest    dc4cda1aa2f1   8 months ago     749MB
final_worker       latest    dc4cda1aa2f1   8 months ago     749MB
flask              latest    58cdeed93a41   9 months ago     448MB
master-web         latest    58cdeed93a41   9 months ago     448MB
creatures          3         503cd4631565   9 months ago     446MB
my_image           manisha   503cd4631565   9 months ago     446MB
homework5          latest    f08364452cbd   9 months ago     490MB
phuong             latest    f08364452cbd   9 months ago     490MB
web_web            latest    a7d00df8fa6a   9 months ago     444MB
web_worker         latest    a7d00df8fa6a   9 months ago     444MB
redis              latest    4cdbec704e47   10 months ago    98.2MB
ubuntu             latest    4e5021d210f6   10 months ago    64.2MB
harmish/gnuplot    latest    b67698a87ae1   2 years ago      392MB
```

Pull an image from Docker hub with the `docker pull` command. This looks through the Docker Hub registry and downloads the 'latest' version of that image:

```
[isp02]$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

Run the image we just pulled with the `docker run` command. In this case, running the container will execute a simple shell script inside the container that has been configured as the 'default command' when the image was built:

```
[isp02]$ docker run hello-world

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

```

(continues on next page)

(continued from previous page)

```
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.
    (amd64)
 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.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Check to see if any containers are still running using `docker ps`:

```
[isp02]$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

EXERCISE

The command `docker ps` shows only currently running containers. Pull up the help text for that command and figure out how to show all containers, not just currently running containers.

3.2.8 Pull Another Image

Navigate to the repositories of user `wallen` on Docker Hub here.

Scroll down to find an image called `wallen/bsd`, then click on that image.

Pull the image using the suggested command, then check to make sure it is available locally:

```
[isp02]$ docker pull wallen/bsd:1.0
...
[isp02]$ docker images
...
[isp02]$ docker inspect wallen/bsd:1.0
...
```

Tip: Use `docker inspect` to find some metadata available for each image.

3.2.9 Start an Interactive Shell Inside a Container

Using an interactive shell is a great way to poke around inside a container and see what is in there. Imagine you are ssh-ing to a different Linux server, have root access, and can see what files, commands, environment, etc., are available.

Before starting an interactive shell inside the container, execute the following commands on the ISP server (we will see why in a minute):

```
[isp02]$ whoami
wallen
[isp02]$ pwd
/home/wallen
[isp02]$ cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (Core)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"

[isp02]$ ls -l /usr/games/
total 0
```

Now start the interactive shell:

```
[isp02]$ docker run --rm -it wallen/bsd:1.0 /bin/bash
root@fc5b620c5a88:/#
```

Here is an explanation of the command options:

```
docker run      # run a container
--rm            # remove the container when we exit
-it             # interactively attach terminal to inside of container
wallen/bsd:1.0  # image and tag on local machine
/bin/bash       # shell to start inside container
```

Try the following commands and note what has changed:

```
root@fc5b620c5a88:/# whoami
root
root@fc5b620c5a88:/# pwd
/
root@fc5b620c5a88:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.6 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.6 LTS"
```

(continues on next page)

(continued from previous page)

```
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
root@fc5b620c5a88:/# ls /usr/games/
adventure   bcd       countmail  hack     morse      ppt     robots   teachgammon␣
↪worms
arithmetic  boggle    cribbage   hangman  number     primes  rot13    tetris-bsd   wtf
atc         caesar    dab        hunt     phantasia  quiz    sail     trek        ␣
↪wump
backgammon  canfield  go-fish    mille    pig        rain    snake    wargames
battlestar  cfscores  gomoku     monop    pom        random  snscore  worm
```

Now you are the `root` user on a different operating system inside a running Linux container! You can type `exit` to escape the container.

EXERCISE

Before you exit the container, try running a few of the games (e.g. `hangman`).

3.2.10 Run a Command Inside a Container

Back out on the ISP server, we now know we have an image called `wallen/bsd:1.0` that has some terminal games inside it which would not otherwise be available to us on the ISP server. They (and their dependencies) are *isolated* from everything else. This image (`wallen/bsd:1.0`) is portable and will run the exact same way on any OS that Docker supports.

In practice, though, we don't want to start interactive shells each time we need to use a software application inside an image. Docker allows you to spin up an *ad hoc* container to run applications from outside. For example, try:

```
[isp02]$ docker run --rm wallen/bsd:1.0 whoami
root
[isp02]$ docker run --rm wallen/bsd:1.0 pwd
/
[isp02]$ docker run --rm wallen/bsd:1.0 cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.6 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.6 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
[isp02]$ docker run --rm wallen/bsd:1.0 ls /usr/games
adventure   bcd       countmail  hack     morse      ppt     robots   teachgammon␣
↪worms
arithmetic  boggle    cribbage   hangman  number     primes  rot13    tetris-bsd   wtf
atc         caesar    dab        hunt     phantasia  quiz    sail     trek        ␣
↪wump
backgammon  canfield  go-fish    mille    pig        rain    snake    wargames
```

(continues on next page)

(continued from previous page)

```
battlestar  cfscores  gomoku     monop    pom        random  snscore  worm
[isp02]$ docker run --rm -it wallen/bsd:1.0 hangman
```

The first four commands above omitted the `-it` flags because they did not require an interactive terminal to run. On each of these commands, Docker finds the image the command refers to, spins up a new container based on that image, executes the given command inside, prints the result, and exits and removes the container.

The last command, which executes the `hangman` game, requires an interactive terminal so the `-it` flags were provided.

3.2.11 Essential Docker Command Summary

Command	Usage
docker login	Authenticate to Docker Hub using username and password
docker images	List images on the local machine
docker ps	List containers on the local machine
docker pull	Download an image from Docker Hub
docker run	Run an instance of an image (a container)
docker inspect	Provide detailed information on Docker objects
docker rmi	Delete an image
docker rm	Delete a container
docker stop	Stop a container
docker build	Build a docker image from a Dockerfile in the current working directory
docker tag	Add a new tag to an image
docker push	Upload an image to Docker Hub

3.2.12 Additional Resources

- Docker Docs
- Best practices for writing Dockerfiles
- Docker Hub
- Docker for Beginners
- Play with Docker

CHAPTER

FOUR

WEEK 4: ADVANCED CONTAINERS, YAML, DOCKER COMPOSE

In this fourth week of class, we will dive deeper into Docker containers by building our own and pushing them to Docker Hub (please make sure you have a Docker Hub account). We will also take a look at the YAML language which is commonly used for container orchestration.

4.1 Advanced Containers

Scenario: You are a developer who has written some new code for reading and parsing objects in JSON format. You now want to distribute that code for others to use in what you know to be a stable production environment (including OS and dependency versions). End users may want to use this application on their local workstations, in the cloud, or on an HPC cluster.

In this section, we will build a Docker image from scratch for running our Python code to parse JSON files. After going through this module, students should be able to:

- Install and test code in a container interactively
- Write a Dockerfile from scratch
- Build a Docker image from a Dockerfile
- Push a Docker image to Docker Hub

4.1.1 Getting Set Up

The first step in a typical container development workflow entails installing and testing an application interactively within a running Docker container.

Note: We recommend doing this on the class ISP server. But, one of the most important features of Docker is that it is platform agnostic. These steps could be done anywhere Docker is installed.

To begin, make a folder for this section and gather the important files. Specifically, you need two files from Homework01: `generate_animals.py` and `read_animals.py`.

```
[isp02]$ cd ~/coe-332/
[isp02]$ mkdir docker-exercise/
[isp02]$ cd docker-exercise/
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
```

Now, we need an empty file called `Dockerfile` and a copy of your previous homework files in here. Optionally, grab a copy of the homework files here:

```
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
[isp02]$ touch Dockerfile
[isp02]$ wget https://raw.githubusercontent.com/tacc/coe-332-sp21/main/docs/week04/
↪scripts/generate_animals.py
[isp02]$ wget https://raw.githubusercontent.com/tacc/coe-332-sp21/main/docs/week04/
↪scripts/read_animals.py
[isp02]$ ls
Dockerfile  generate_animals.py  read_animals.py
```

Warning: It is important to carefully consider what files and folders are in the same `PATH` as a Dockerfile (known as the 'build context'). The `docker build` process will index and send all files and folders in the same directory as the Dockerfile to the Docker daemon, so take care not to `docker build` at a root level.

4.1.2 Containerize Code Interactively

There are several questions you must ask yourself when preparing to containerize code for the first time:

1. What is an appropriate base image?
2. What dependencies are required for my program?
3. What is the install process for my program?
4. What environment variables may be important?

We can work through these questions by performing an **interactive installation** of our Python scripts. Our development environment (the class ISP server) is a Linux server running CentOS 7.7. We know our code works here, so that is how we will containerize it. Use `docker run` to interactively attach to a fresh CentOS 7.7 container.

```
[isp02]$ docker run --rm -it -v $PWD:/code centos:7.7.1908 /bin/bash
[root@fbf511fa3447 /]#
```

Here is an explanation of the options:

```
docker run       # run a container
--rm             # remove the container on exit
-it              # interactively attach terminal to inside of container
-v $PWD:/code    # mount the current directory to /code
centos:7.7.1908  # image and tag from Docker Hub
/bin/bash        # shell to start inside container
```

The command prompt will change, signaling you are now 'inside' the container. And, new to this example, we are using the `-v` flag which mounts the contents of our current directory (`$PWD`) inside the container in a folder in the root directory called (`/code`).

Update and Upgrade

The first thing we will typically do is use the CentOS package manager `yum` to update the list of available packages and install newer versions of the packages we have. We can do this with:

```
[root@fbf511fa3447 /]# yum update
...
```

Note: You will need to press 'y' followed by 'Enter' twice to download and install the updates

Install Required Packages

For our Python scripts to work, we need to install two dependencies: Python3 and the 'petname' package.

```
[root@fbf511fa3447 /]# yum install python3
...
[root@fbf511fa3447 /]# python3 --version
Python 3.6.8
[root@fbf511fa3447 /]# pip3 install petname==2.6
...
 UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position 32: ordinal not␣
↪in range(128)
...
 Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-build-
↪8qhc6nih/petname/
```

Oh no! A `UnicodeDecodeError` happens sometimes on pip installs. We can fix this by manually setting the character encoding with a couple environment variables and trying again.

```
[root@fbf511fa3447 /]# export LC_CTYPE=en_US.UTF-8
[root@fbf511fa3447 /]# export LANG=en_US.UTF-8
[root@fbf511fa3447 /]# pip3 install petname
...
  Successfully installed petname-2.6
```

Success! Now all of our dependencies are installed and we can move on to the JSON parsing code.

Warning: An important question to ask is: Does the versions of Python and other dependencies match the versions you are developing with in your local environment? If not, make sure to install the correct version of Python.

Install and Test Your Code

At this time, we should make a few small edits to the code that will make them a lot more amenable to running in a container. There are two specific changes. First, add a 'shebang' to the top of your scripts to make them executable without calling the Python3 interpreter:

```
#!/usr/bin/env python3
```

Second, instead of hard coding the filename 'animals.json' in each script, let's make a slight modification so we can pass the filename on the command line. In each script, add this line near the top:

```
import sys
```

And change the `with open...` statements to these, as appropriate:

```
with open(sys.argv[1], 'w') as f:           # in generate_animals.py
    json.dump(animal_dict, f, indent=2)     #

with open(sys.argv[1], 'r') as f:           # in read_animals.py
    animal_dict = json.load(f)              #
```

Tip: If you are using the sample files linked above, they already have these changes in them.

Since we are using simple Python scripts, there is not a difficult install process. However, we can make them executable and add them to them user's *PATH*.

```
[root@fbf511fa3447 /]# cd /code
[root@fbf511fa3447 /]# chmod +rx generate_animals.py
[root@fbf511fa3447 /]# chmod +rx read_animals.py
[root@fbf511fa3447 /]# export PATH=/code:$PATH
```

Now test with the following:

```
[root@fbf511fa3447 /]# cd /home
[root@fbf511fa3447 /]# generate_animals.py animals.json
[root@fbf511fa3447 /]# ls
animals.json
[root@fbf511fa3447 /]# read_animals.py animals.json
{'head': 'bunny', 'body': 'yeti-ibex', 'arms': 8, 'legs': 12, 'tail': 20}
```

We now have functional versions of both scripts 'installed' in this container. Now would be a good time to execute the *history* command to see a record of the build process. When you are ready, type *exit* to exit the container and we can start writing these build steps into a Dockerfile.

4.1.3 Assemble a Dockerfile

After going through the build process interactively, we can translate our build steps into a Dockerfile using the directives described below. Open up your copy of `Dockerfile` with a text editor and enter the following:

The FROM Instruction

We can use the FROM instruction to start our new image from a known base image. This should be the first line of our Dockerfile. In our scenario, we want to match our development environment with CentOS 7.7. We know our code works in that environment, so that is how we will containerize it for others to use:

```
FROM centos:7.7.1908
```

Base images typically take the form *os:version*. Avoid using the '*latest*' version; it is hard to track where it came from and the identity of '*latest*' can change.

Tip: Browse Docker Hub to discover other potentially useful base images. Keep an eye out for the 'Official Image' badge.

The RUN Instruction

We can install updates, install new software, or download code to our image by running commands with the RUN instruction. In our case, our only dependencies were Python3 and petname. So, we will use a few RUN instructions to install them. Keep in mind that the the `docker build` process cannot handle interactive prompts, so we use the `-y` flag with `yum` and `pip3`.

```
RUN yum update -y
RUN yum install -y python3
RUN pip3 install petname==2.6
```

Each RUN instruction creates an intermediate image (called a 'layer'). Too many layers makes the Docker image less performant, and makes building less efficient. We can minimize the number of layers by combining RUN instructions:

```
RUN yum update -y && yum install -y python3
RUN pip3 install petname==2.6
```

The COPY Instruction

There are a couple different ways to get your source code inside the image. One way is to use a RUN instruction with `wget` to pull your code from the web. When you are developing, however, it is usually more practical to copy code in from the Docker build context using the COPY instruction. For example, we can copy our scripts to the root-level */code* directory with the following instructions:

```
COPY generate_animals.py /code/generate_animals.py
COPY read_animals.py /code/read_animals.py
```

And, don't forget to perform two more RUN instruction to make the scripts executable:

```
RUN chmod +rx /code/generate_animals.py && \
    chmod +rx /code/read_animals.py
```

Tip: In the above code block, the `\` character at the end of the lines causes the newline character to be ignored. This can make very long run-on lines with many commands separated by `&&` easier to read.

The ENV Instruction

Another useful instruction is the ENV instruction. This allows the image developer to set environment variables inside the container runtime. In our interactive build, we added the `/code` folder to the `PATH`, and we also had to set a few environment variables for the character set. We can do this with ENV instructions as follows:

```
ENV LC_CTYPE=en_US.UTF-8
ENV LANG=en_US.UTF-8

ENV PATH "/code:$PATH"
```

Warning: Be mindful where these instructions appear in your Dockerfile! The encoding environment variables must appear in the file before they are needed.

Putting It All Together

The contents of the final Dockerfile should look like:

```
FROM centos:7.7.1908

RUN yum update -y && yum install -y python3

ENV LC_CTYPE=en_US.UTF-8
ENV LANG=en_US.UTF-8

RUN pip3 install petname==2.6

COPY generate_animals.py /code/generate_animals.py
COPY read_animals.py /code/read_animals.py

RUN chmod +rx /code/generate_animals.py && \
    chmod +rx /code/read_animals.py

ENV PATH "/code:$PATH"
```

4.1.4 Build the Image

Once the Dockerfile is written and we are satisfied that we have minimized the number of layers, the next step is to build an image. Building a Docker image generally takes the form:

```
[isp02]$ docker build -t <dockerhubusername>/<code>:<version> .
```

The `-t` flag is used to name or 'tag' the image with a descriptive name and version. Optionally, you can preface the tag with your **Docker Hub username**. Adding that namespace allows you to push your image to a public registry and share it with others. The trailing dot '`.`' in the line above simply indicates the location of the Dockerfile (a single '`.`' means 'the current directory').

To build the image, use:

```
[isp02]$ docker build -t username/json-parser:1.0 .
```

Note: Don't forget to replace 'username' with your Docker Hub username.

Use `docker images` to ensure you see a copy of your image has been built. You can also use *docker inspect* to find out more information about the image.

```
[isp02]$ docker images
REPOSITORY           TAG        IMAGE ID       CREATED          SIZE
wallen/json-parser   1.0        632f9f174274   33 minutes ago   507MB
...
centos               7.7.1908   08d05d1d5859   15 months ago    204MB
```

```
[isp02]$ docker inspect username/json-parser:1.0
```

If you need to rename your image, you can either re-tag it with `docker tag`, or you can remove it with `docker rmi` and build it again. Issue each of the commands on an empty command line to find out usage information.

4.1.5 Test the Image

We can test a newly-built image two ways: interactively and non-interactively. In interactive testing, we will use `docker run` to start a shell inside the image, just like we did when we were building it interactively. The difference this time is that we are NOT mounting the code inside with the `-v` flag, because the code is already in the container:

```
[isp02]$ docker run --rm -it username/json-parser:1.0 /bin/bash
...
[root@c5cf05edddcd /]# ls /code
generate_animals.py  read_animals.py
[root@c5cf05edddcd /]# cd /home
[root@c5cf05edddcd home]# pwd
/home
[root@c5cf05edddcd home]# generate_animals.py test.json
[root@c5cf05edddcd home]# ls
test.json
[root@c5cf05edddcd home]# read_animals.py test.json
{'head': 'snake', 'body': 'camel-oyster', 'arms': 8, 'legs': 12, 'tail': 20}
```

Here is an explanation of the options:

```
docker run      # run a container
--rm            # remove the container when we exit
-it             # interactively attach terminal to inside of container
username/...    # image and tag on local machine
/bin/bash       # shell to start inside container
```

Everything looks like it works! Next, exit the container and test the code non-interactively. Notice we are calling the container again with `docker run`, but instead of specifying an interactive (`-it`) run, we just issue the command as we want to call it on the command line. Also, notice the return of the `-v` flag, because we need to create a volume mount so that our data (`animals.json`) will persist outside the container.

```
[isp02]$ mkdir test
[isp02]$ cd test
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise/test
[isp02]$ docker run --rm -v $PWD:/data username/json-parser:1.0 generate_animals.py /
↪data/animals.json
[isp02]$ ls -l
total 4
-rw-r--r-- 1 root root 2325 Feb  8 14:30 animals.json
```

A new file appeared! The file `animals.json` was written by a the Python script inside the container, and because we mounted our current location as a folder called '/data' (`-v $PWD:/data`), and we made sure to write the output file to that location in the container (`generate_animals.py /data/animals.json`), then we get to keep the file.

Alas, there is an issue. The new file is owned by `root`, simply because it is `root` who created the file inside the container. This is one minor Docker annoyance that we run in to from time to time. The simplest fix is to use one more `docker run` flag to specify the user and group ID namespace that should be used inside the container.

```
[isp02]$ rm animals.json
rm: remove write-protected regular file 'animals.json'? y
[isp02]$ docker run --rm -v $PWD:/data -u $(id -u):$(id -g) username/json-parser:1.0␣
↪generate_animals.py /data/animals.json
[isp02]$ ls -l
total 4
-rw-r--r-- 1 wallen G-815499 2317 Feb  8 14:40 animals.json
```

Much better! And finally, we can test the `read_animals.py` script on the file we just created:

```
[isp02]$ docker run --rm -v $PWD:/data username/json-parser:1.0 read_animals.py /data/
↪animals.json
{'head': 'lion', 'body': 'rhino-duck', 'arms': 6, 'legs': 12, 'tail': 18}
```

This time, we still mount the volume with `-v` so that the `read_animals.py` script has access to the input file inside the container. But we don't use the `-u` flag because we are not writing any new files and user namespace does not need to be enforced.

4.1.6 Share Your Docker Image

Now that you have containerized, tested, and tagged your code in a Docker image, the next step is to disseminate it so others can use it.

Docker Hub is the *de facto* place to share an image you built. Remember, the image must be name-spaced with either your Docker Hub username or a Docker Hub organization where you have write privileges in order to push it:

```
[isp02]$ docker login
...
[isp02]$ docker push username/json-parser:1.0
```

You and others will now be able to pull a copy of your container with:

```
[isp02]$ docker pull username/json-parser:1.0
```

As a matter of best practice, it is highly recommended that you store your Dockerfiles somewhere safe. A great place to do this is alongside the code in, e.g., GitHub. GitHub also has integrations to automatically update your image in the public container registry every time you commit new code.

For example, see: Set up automated builds

4.1.7 Additional Resources

- Docker for Beginners
- Play with Docker

4.2 YAML Reference

This reference guide is designed to introduce you to YAML syntax. YAML is basically JSON with a couple extra features, and meant to be a little more human readable. We will be using YAML formatted configuration files in the Docker compose and Kubernetes sections, so it is important to become familiar with the syntax.

4.2.1 YAML Basics

YAML syntax is similar to Python dictionaries, and we will usually see them as key:value pairs. Values can include strings, numbers, booleans, null, lists, and other dictionaries.

Previously, we saw a simple JSON object in dictionary form like:

```
{
  "key1": "value1",
  "key2": "value2"
}
```

That same object in YAML looks like:

```
---
key1: value1
key2: value2
...
```

Notice that YAML documents all start with three hyphens on the top line (`---`), and end with an optional three dots (`...`) on the last line. Key:value pairs are separated by colons, but consecutive key:value pairs are NOT separated by commas.

We also mentioned that JSON supports list-like structures. YAML does too. So the following valid JSON block:

```
[
  "thing1", "thing2", "thing3"
]
```

Appears like this in YAML:

```
---
- thing1
- thing2
- thing3
...
```

Elements of the same list all appear with a hyphen `-` at the same indent level.

We also previously saw this complex data structure in JSON:

```
{
  "department": "COE",
  "number": 332,
  "name": "Software Engineering and Design",
  "inperson": false,
  "instructors": ["Joe", "Charlie", "Brandi", "Joe"],
  "prerequisites": [
    {"course": "COE 322", "instructor": "Victor"},
    {"course": "SDS 322", "instructor": "Victor"}
  ]
}
```

The same structure in YAML is:

```
---
department: COE
number: 332
```

(continues on next page)

(continued from previous page)

```yaml
name: Software Engineering and Design
inperson: false
instructors:
  - Joe
  - Charlie
  - Brandi
  - Joe
prerequisites:
  - course: COE 322
    instructor: Victor
  - course: SDS 322
    instructor: Victor
...
```

The whole thing can be considered a dictionary. The key `instructors` contains a value that is a list of names, and the key `prerequisites` contains a value that is a list of two dictionaries. Booleans appear as `false` and `true` (lowercase only). A null / empty value would appear as `null`.

Also, check out the list of states we worked with in the JSON section, but now in YAML here.

4.2.2 More YAML

There is a lot more to YAML, most of which we will not use in this course. Just know that YAML files can contain:

- Comments
- Multi-line strings / text blocks
- Multi-word keys
- Complex objects
- Special characters
- Explicitly declared types
- A mechanism to duplicate / inherit values across a document ("anchors")

If we encounter a need for any of these, we can refer to the official YAML syntax

4.2.3 Additional Resources

- YAML Spec
- YAML Validator
- JSON / YAML Converter

4.3 Docker Compose

Up to this point, we have been looking at single-container applications - small units of code that are containerized, executed *ad hoc* to generate or read a JSON file, then exit on completion. But what if we want to do something more complex? For example, what if our goal is to orchestrate a multi-container application consisting of, e.g., a Flask app, a database, a message queue, an authentication service, and more.

Docker compose is tool for managing multi-container applications. A YAML file is used to define all of the application service, and a few simple commands can be used to spin up or tear down all of the services.

In this module, we will get a first look at Docker compose. Later in this course we will do a deeper dive into advanced container orchestration. After going through this module, students should be able to:

- Translate Docker run commands into YAML files for Docker compose
- Run commands inside *ad hoc* containers using Docker compose

4.3.1 A Simple Example

Docker compose works by interpreting rules declared in a YAML file (typically called `docker-compose.yml`). The rules we will write will replace the `docker run` commands we have been using, and which have been growing quite complex. For example, the commands we used to run our JSON parsing scripts in a container looked like the following:

```
[isp02]$ docker run --rm -v $PWD:/data -u $(id -u):$(id -g) username/json-parser:1.0␣
↪generate_animals.py /data/animals.json
[isp02]$ docker run --rm -v $PWD:/data -u $(id -u):$(id -g) username/json-parser:1.0␣
↪read_animals.py /data/animals.json
```

The above `docker run` commands can be loosely translated into a YAML file. Navigate to the folder that contains your Python scripts and Dockerfile, then create a new empty file called `docker-compose.yml`:

```
[isp02]$ pwd
/home/wallen/coe-332/docker-exercise
[isp02]$ touch docker-compose.yml
[isp02]$ ls -l
total 12
-rw-------. 1 wallen G-815499   0 Feb 10 20:46 docker-compose.yml
-rw-------. 1 wallen G-815499 329 Feb  9 12:39 Dockerfile
-rw-------. 1 wallen G-815499 703 Feb  9 11:16 generate_animals.py
-rw-------. 1 wallen G-815499 236 Feb  9 11:16 read_animals.py
drwx------. 2 wallen G-815499   6 Feb 10 20:44 test/
```

Next, open up `docker-compose.yml` with your favorite text editor and type / paste in the following text:

```
---
version: "3"

services:
    gen-anim:
        image: wallen/json-parser:1.0
        volumes:
            - ./test:/data
        user: "827385:815499"
        command: generate_animals.py /data/animals.json
    read-anim:
```

(continues on next page)

(continued from previous page)

```
        image: wallen/json-parser:1.0
        volumes:
            - ./test:/data
        user: "827385:815499"
        command: read_animals.py /data/animals.json
...
```

The `version` key must be included and simply denotes that we are using version 3 of Docker compose.

The `services` section defines the configuration of individual container instances that we want to orchestrate. In our case, we define two called `gen-anim` for the generate_animals functionality, and `read-anim` for the read_animals functionality.

Each of those services is configured with a Docker image (`wallen/json-parser:1.0`), a mounted volume (equivalent to the `-v` option for `docker run`), a user namespace (equivalent to the `-u` option for `docker run`), and a default command to run.

Please note that the image name above should be changed to use your image. Also, the user ID / group ID are specific to `wallen` - to find your user and group ID, execute the Linux commands `id -u` and `id -g`.

Note: The top-level `services` keyword shown above is just one important part of Docker compose. Later in this course we will look at named volumes and networks which can be configured and created with Docker compose.

4.3.2 Running Docker Compose

The Docker compose command line too follows the same syntax as other Docker commands:

```
docker-compose <verb> <parameters>
```

Just like Docker, you can pass the `--help` flag to `docker-compose` or to any of the verbs to get additional usage information. To get started on the command line tools, try issuing the following two commands:

```
[isp02]$ docker-compose version
[isp02]$ docker-compose config
```

The first command prints the version of Docker compose installed, and the second searches your current directory for `docker-compose.yml` and checks that it contains only valid syntax.

To run one of these services, use the `docker-compose run` verb, and pass the name of the service as defined in your YAML file:

```
[isp02]$ ls test/      # currently empty
[isp02]$ docker-compose run gen-anim
[isp02]$ ls test/
animals.json           # new file!
[isp02]$ docker-compose run read-anim
{'head': 'snake', 'body': 'marlin-tapir', 'arms': 10, 'legs': 9, 'tail': 19}
```

Now we have an easy way to run our *ad hoc* services consistently and reproducibly. Not only does `docker-compose.yml` make it easier to run our services, it also represents a record of how we intend to interact with this container.

4.3.3 Essential Docker Compose Command Summary

Command	Usage
docker-compose version	Print version information
docker-compose config	Validate docker-compose.yml syntax
docker-compose up	Spin up all services
docker-compose down	Tear down all services
docker-compose build	Build the images listed in the YAML file
docker-compose run	Run a container as defined in the YAML file

4.3.4 Additional Resources

- Docker Compose Docs

CHAPTER

FIVE

WEEK 6: INTRO TO APIS, INTRO TO FLASK

In this sixth week of class, we will be introduced to Application Programming Interfaces (APIs). This introduction will form the foundation of our ultimate goal to create large, complex, python-based applications that are accessible through the web. The particular REST API framework we will be working with most is called Flask.

5.1 Introduction to APIs

An Application Programming Interface (API) establishes the protocols and methods for one piece of a program to communicate with another. APIs are useful for (1) allowing larger software systems to be built from smaller components, (2) allowing the same component/code to be used by different systems, and (3) insulating consumers from changes to the implementation.

Some examples of APIs:

- In OOP languages, abstract classes provide the interface for all concrete classes to implement
- Software libraries provide an external interface for consuming programs
- Web APIs (or "web services") provide interfaces for computer programs to communicate over the internet

In this section, we will see some Web APIs, and in particular REST APIs, and we will learn how to interact with them using Python scripts. After going through this module, students should be able to:

- Identify and describe Web APIs (including REST APIs)
- Find API endpoints to various websites, e.g. Bitbucket and GitHub
- List and define the four most important HTTP verbs
- Install and import the Python requests library
- Interact with a web API using the Python requests library, and parse the return information

5.1.1 Web APIs

In this course, we will focus on Web APIs (or HTTP APIs). These are interfaces that are exposed over HTTP. There are a number of advantages to Web-based APIs that we will use in this class:

- A Web API is accessible from any computer or application that has access to the public internet
- No software installation is required on the client's side to consume a web API
- Web APIs can change their implementation without clients knowing (or caring)
- Virtually every modern programming language provides one or more libraries for interacting with a web API - thus, "language agnostic"

5.1.2 HTTP - the Protocol of the Internet

HTTP (Hyper Text Transfer Protocol) is one way for two computers on the internet to communicate with each other. It was designed to enable the exchange of data (specifically, "hypertext"). In particular, our web browsers use HTTP when communicating with web servers running web applications. HTTP uses a message-based, **client-server model**: clients make requests to servers by sending a message, and servers respond by sending a message back to the client.

HTTP is an "application layer" protocol in the language of the Internet Protocols; it assumes a lower level transport layer protocol. While this can swapped, in practice it is almost always TCP. The basics of the protocol are:

- Web resources are identified with URLs (Uniform Resource Locators). Originally, **resources** were just files/directories on a server, but today resources refer to more general objects.
- HTTP "verbs" represent actions to take on the resource. The most common verbs are `GET`, `POST`, `PUT`, and `DELETE`.
- A **request** is made up of a URL, an HTTP verb, and a message
- A **response** consists of a status code (numerical between 100-599) and a message. The first digit of the status code specifies the kind of response:
  - 1xx - informational
  - 2xx - success
  - 3xx - redirection
  - 4xx - error in the request (client)
  - 5xx - error fulfilling a valid request (server)

5.1.3 REST APIs - Overview

REST (Representational State Transfer) is a way of building APIs for computer programs on the internet leveraging HTTP. In other words, a program on computer 1 interacts with a program on computer 2 by making an HTTP request to it.

In HTTP terms, "resources" are the nouns of the application domain and are associated with URLs. The API has a **base URL** from which all other URLs in that API are formed, e.g.:

```
https://api.github.com/
```

The other URLS in the API are either "collections", e.g.:

```
<base_url>/users
<base_url>/files
<base_url>/programs
```

or they are specific items in a collection, e.g.:

```
<base_url>/users/12345
<base_url>/files/test.txt
<base_url>/programs/myapplication
```

or subcollections / items in subcollections, e.g.:

```
<base_url>/companies/<company_id>/employees
<base_url>/companies/<company_id>/employees/<employee_id>
```

Continuing along with HTTP terms, "operations" are the actions that can be taken on the resources and are associated with HTTP verbs:

- `GET` - list items in a collection or retrieve a specific item in the collection
- `POST` - create a new item in the collection based on the description in the message
- `PUT` - replace an item in a collection with the description in the message
- `DELETE` - delete an item in a collection

Response messages often make use of some data serialization format standard such as `JSON` or `XML`.

Note: The base URL to the GitHub API is listed above, https://api.github.com/. You can discover the API to GitHub and other popular websites by searching in Google something like "GitHub API endpoint".

5.1.4 REST APIs - Toy Examples

Virtually every application domain can be mapped into a REST API architecture. Some examples may include:

Articles in a collection (e.g., on a blog or wiki) with author attributes:

```
<base_url>/articles
<base_url>/articles/<id>
<base_url>/articles/<id>/authors
```

Properties in a real estate database with associated purchase history:

```
<base_url>/properties
<base_url>/properties/<id>
<base_url>/properties/<id>/purchases
```

A catalog of countries, cities and neighborhoods:

```
<base_url>/countries
<base_url>/countries/<country_id>/cities
<base_url>/countries/<country_id>/cities/<city_id>/neighborhoods
```

5.1.5 REST APIs - A Real Example

Bitbucket is a website for managing git repositories. You may already be familiar with the Bitbucket website, https://bitbucket.org/. Let's now take a look at the Bitbucket Web API. Open a web browser and navigate to:

- https://api.bitbucket.org/2.0/repositories

When you opened that page, your browser made a `GET` request to the Bitbucket API. What you see is a `JSON` object describing public repositories.

If you look closely, you will see three top level objects in the response: `pagelen` (int), `values` (list), and `next` (str). What do you think each represents?

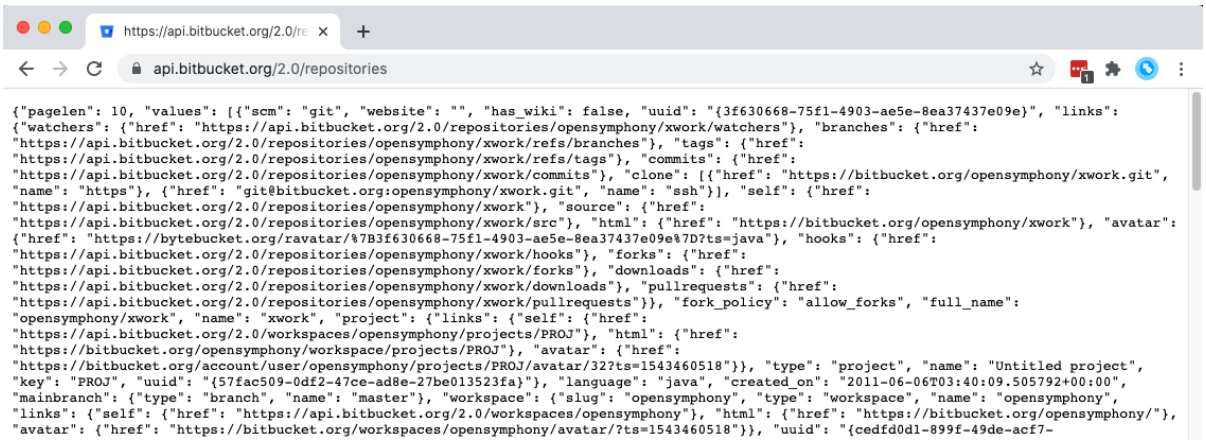


Fig. 1: The first entries returned from the Bitbucket API.

EXERCISE

- Were all Bitbucket repositories returned? How many were returned? What URL would you use to get the next set of repositories?
- What URL would we use to get a list of public repositories owned by a specific user?
- What URL would we use to get a list of commits for a specific public repository?

Tip: Web APIs for popular sites (like Bitbucket) often come with online documentation.

5.1.6 Using Python to Interact with Web APIs

Viewing API response messages in a web browsers is of limited use. We can interact with Web APIs in a much more powerful and programmatic way using the Python `requests` library.

First install the `requests` library in your userspace on the ISP server using pip:

```
[isp02]$ pip3 install --user requests
...
Successfully installed requests-2.25.1
```

You might test that the install was successful by trying to import the library in the interactive Python interpreter:

```
[isp02]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```

The basic usage of the `requests` library is as follows:

```
>>> # make a request
>>> response = requests.<method>(url=some_url, data=some_message, <other options>)
>>>
>>> # e.g. try:
```

(continues on next page)

(continued from previous page)

```
>>> response = requests.get(url='https://api.bitbucket.org/2.0/repositories')
>>>
>>> # return the status code:
>>> response.status_code
>>>
>>> # return the raw content
>>> response.content
>>>
>>> # return a Python list or dictionary from the response message
>>> response.json()
```

EXERCISE

Let's explore the Bitbucket API using the `requests` library in a Python script. Write functions to return the following:

- Retrieve a list of public bitbucket repositories
- Retrieve a list of public bitbucket repositories for a particular user
- Retrieve a list of pull requests for a particular public bitbucket repository

5.1.7 Additional Resources

- Bitbucket API
- Bitbucket API Documentation
- Python requests Documentation

5.2 Introduction to Flask

Flask is a Python library and framework for building web servers. Some of the defining characteristics of Flask make it a good fit for this project:

- Flask is small and lightweight - relatively easy to use and get setup initially
- Flask is robust - a great fit for REST APIs and **microservices**
- Flask is performant - when used correctly, it can handle the traffic of sites with 100Ks of users

In this section, we will get a brief introduction into Flask, including how to set up a quick REST API with multiple routes (URLs). After going through this module, students should be able to:

- Install the Flask Python library and import it into a Python script
- Define and give function to various routes in a Flask Python script
- Run a local Flask development server
- Curl defined routes from the local Flask server

5.2.1 Wait - What is a Microservice?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack. Many heavily-trafficked, well-known sites use microservices including Netflix, Amazon, and eBay.

There is a great article on DevTeam.Space about microservices.

5.2.2 Setup and Installation

The Flask library is not part of the Python standard library but can be installed standard tools like `pip3`. In addition to making Flask available to import into a Python script, it will also expose some new command line tools. On the class server, perform the following:

```
[isp02]$ pip install --user flask
...
Successfully installed flask-1.1.2

[isp02]$ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...

A general utility script for Flask applications.

Provides commands from Flask, extensions, and the application. Loads the
application defined in the FLASK_APP environment variable, or from a
wsgi.py file. Setting the FLASK_ENV environment variable to 'development'
will enable debug mode.

  > export FLASK_APP=hello.py
  > export FLASK_ENV=development
  > flask run

Options:
  --version  Show the flask version
  --help     Show this message and exit.

Commands:
  routes  Show the routes for the app.
  run     Run a development server.
  shell   Run a shell in the app context.
```

5.2.3 A Hello World Flask App

In a new directory on the class server, create a file called `app.py` and open it for editing. Enter the following lines of code:

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  # the next statement should usually appear at the bottom of a flask app
6  if __name__ == '__main__':
7      app.run(debug=True, host='0.0.0.0')
```

On the first line, we are importing the Flask class.

On the third line, we create an instance of the Flask class (called `app`). This so-called "Flask application" object holds the primary configuration and behaviors of the web program.

Finally, the `app.run()` method launches the development server. The `debug=True` option tells Flask to print verbose debug statements while the server is running. The `host=0.0.0.0` option instructs the server to listen on all network interfaces; basically this means you can reach the server from inside and outside the host VM.

5.2.4 Run the Flask App

There are two main ways of starting the Flask service. For now, we recommend you start the service using a unique port number. The `-p 5000` indicates that Flask is running on port 5000. You will need to use your own assigned port.

Warning: Check Slack or ask the instructors which port you should use. Trying to run two Flask apps on the same port will not work.

```
[isp02]$ export FLASK_APP=app.py
[isp02]$ export FLASK_ENV=development
[isp02]$ flask run -p 5000
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 182-299-771
```

That's it! We now have a server up and running. Some notes on what is happening:

- Note that the program took over our shell; we could put it in the background, but for now we want to leave it in the foreground. (Multiple PIDs are started for the Flask app when started in daemon mode; to get them, find all processes listening on the port 5000 socket with `lsof -i:5000`).
- If we make changes to our Flask app while the server is running in development mode, the server will detect those changes automatically and "reload"; you will see a log to the effect of `Detected change in <file>`.
- We can stop the program with `Ctrl+C` just like any other (Python) program.
- If we stop our Flask programs, the server will no longer be listening and our requests will fail.

Next we can try to talk to the server using `curl`. Note this line:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

That tells us our server is listening on the `localhost` - `127.0.0.1`, and on the default Flask port, port `5000`.

Ports Basics

Ports are a concept from networking that allow multiple services or programs to be running at the same time, listening for messages over the internet, on the same computer.

- For us, ports will always be associated with a specific IP address. In general, we specify a port by combining it with an IP separated by a colon (:) character. For example, `129.114.97.16:5000`.
- One and only one program can be listening on a given port at a time.
- Some ports are designated for specific activities; Port 80 is reserved for HTTP, port 443 for HTTPS (encrypted HTTP), but other ports can be used for HTTP/HTTPS traffic.

curl Basics

You can think of `curl` as a command-line version of a web browser: it is just an HTTP client.

- The basic syntax is `curl <some_url>:<some_port>`. This will make a `GET` request to the URL and port print the message response.
- Curl will default to using port 80 for HTTP and port 443 for HTTPS.
- You can specify the HTTP verb to use with the `-X` flag; e.g., `curl -X GET <some_url>` (though `-X GET` is redundant because that is the default mode).
- You can set "verbose mode" with the `-v` flag, which will then show additional information such as the headers passed back and forth (more on this later).

5.2.5 Make a Request

Because the terminal window running your Flask app is currently locked to that process, the simplest thing to do is open up a new terminal and SSH into the class server again.

To make a request to your Flask app, type the following in the new terminal:

```
[isp02]$ curl 127.0.0.1:5000
- or -
[isp02]$ curl localhost:5000
```

You should see the following response:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.  If you entered the URL manually␣
↪please check your spelling and try again.</p>
```

Our server is sending us HTML! It's sending a 404 that it could not find the resource we requested. Although it appears to be an error (and technically it is), this is evidence that the Flask server is running successfully. It's time to add some routes.

Note: Only one Flask app can be associated with each port. The default port above (5000) is an example. Please make sure to run your Flask server on the port assigned to you (`flask run -p 50xx`). You can curl your own port number, or you can curl other people's Flask servers by subbing in their port number.

5.2.6 Routes in Flask

In a Flask app, you define the URLs in your application using the `@app.route` decorator. Specifications of the `@app.route` decorator include:

- Must be placed on the line before the declaration of a Python function.
- Requires a string argument which is the path of the URL (not including the base URL)
- Takes an argument `methods` which should be a list of strings containing the names of valid HTTP methods (e.g. `GET`, `POST`, `PUT`, `DELETE`)

When the URL + HTTP method combination is requested, Flask will call the decorated function.

Tangent: What is a Python Decorator?

A decorator is a function that takes another function as an input and extends its behavior in some way. The decorator function itself must return a function which includes a call to the original function plus the extended behavior. The typical structure of a decorator is as follows:

```
def my_decorator(some_func):

    def func_to_return():

        # extend the behavior of some_func by doing some processing
        # before it is called (optional)
        do_something_before()

        # call the original function
        some_func(*args, **kwargs)

        # extend the behavior of some_func by doing some processing
        # after it is called (optional)
        do_something_after()

    return func_to_return
```

As an example, consider this test program:

```
def print_dec(f):
    def func_to_return(*args, **kwargs):
        print("args: {}; kwargs: {}".format(args, kwargs))
        val = f(*args, **kwargs)
        print("return: {}".format(val))
        return val
    return func_to_return

@print_dec
def foo(a):
    return a+1
```

(continues on next page)

(continued from previous page)

```
12
13 result = foo(2)
14 print("Got the result: {}".format(result))
```

Our `@print_dec` decorator gets executed automatically when we call `foo(2)`. Without the decorator, the final output would be:

```
Got the result: 3
```

By using the decorator, however, the final output is instead:

```
args: (2,); kwargs: {}
return: 3
Got the result: 3
```

5.2.7 Define the Hello World Route

The original Flask app we wrote above (in `app.py`) did not define any routes. Let's define a "hello world" route for the base URL. Meaning if someone were to curl against the base URL (`/`) of our server, we would want to return the message "Hello, world!". To do so, add the following lines to your `app.py` script:

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/', methods=['GET'])
6  def hello_world():
7      return 'Hello, world!\n'
8
9  # the next statement should usually appear at the bottom of a flask app
10 if __name__ == '__main__':
11     app.run(debug=True, host='0.0.0.0')
```

The `@app.route` decorator on line 5 is expecting `GET` requests at the base URL `/`. When it receives such a request, it will execute the `hello_world()` function below it.

In your active SSH terminal, execute the curl command again (you may need to restart the Flask app); you should see:

```
[isp02]$ curl localhost:5000/
Hello, world!
```

5.2.8 Routes with URL Parameters

Flask makes it easy to create Routes (or URLs) with variables in the URL. The variable name simply must appear in angled brackets (<>) within the `@app.route()` decorator statement; for example:

```
@app.route('/<year>')
```

Would grant the function below it access to a variable called `year`

In the following example, we extend our `app.py` Flask app by adding a route with a variable (`<name>`):

```python
from flask import Flask

app = Flask(__name__)

@app.route('/', methods=['GET'])
def hello_world():
    return 'Hello, world!\n'

@app.route('/<name>', methods=['GET'])
def hello_name(name):
    return f'Hello, {name}!\n'

# the next statement should usually appear at the bottom of a flask app
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Now, the Flask app supports multiple routes with different functionalities:

```
[isp02]$ curl localhost:5000/
Hello, world!
[isp02]$ curl localhost:5000/joe
Hello, joe!
[isp02]$ curl localhost:5000/jane
Hello, jane!
```

EXERCISE

Note: This exercise will be reflected in Homework 03, parts A and B.

Using your creature creator dataset, use your get_data() function that reads in your data set into a dictionary.

```python
def get_data():
    ....
```

You job is to create an API to manage that database. We need to think through the following:

- What are the nouns in our application?
- What are the routes we want to define?
- What data format do we want to return?

Part A: Create some new `GET` routes for the nouns identified in the database above. Find your nouns, make at least 3 routes to retrieve the nouns from your JSON data.

Part B: Write tests for your routes.

5.2.9 Additional Resources

- Python Virtual Environments
- Flask Documentation

CHAPTER

SIX

WEEK 7: ADVANCED FLASK, CONTAINERIZING FLASK

In this seventh week of class, we will dive deeper into the Flask framework to see how it can employ multiple URLs / endpoints, receive and decode JSON data, and handle query parameters. We will also containerize a Flask app using Docker.

6.1 Advanced Flask

Following our brief introduction to the Flask framework, we continue here with a look at more complex endpoints and data retrieval functions for our REST API. After going through this module, students should be able to:

- Identify valid and invalid Flask route return types
- Convert unsupported types (e.g. `list`) to valid Flask route return types
- Extract Content-Type and other headers from Flask route responses
- Add query parameters to GET requests, and extract their values inside Flask routes

6.1.1 Defining the URLs of Our API

The first basic goal of our API is to provide an interface to a dataset. Since the URLs in a REST API are defined by the "nouns" or collections of the application domain, we can use a noun that represents our data.

For example, suppose we have the following dataset that represents the number of students earning an undergraduate degree for a given year:

```
def get_data():
    return [ {'id': 0, 'year': 1990, 'degrees': 5818},
    {'id': 1, 'year': 1991, 'degrees': 5725},
    {'id': 2, 'year': 1992, 'degrees': 6005},
    {'id': 3, 'year': 1993, 'degrees': 6123},
    {'id': 4, 'year': 1994, 'degrees': 6096} ]
```

In this case, one collection described by the data is "degrees". So, let's define a route, `/degrees`, that by default returns all of the data points.

EXERCISE 1

Create a new file, `degrees_api.py` to hold a Flask application then do the following:

- Import the Flask class and instantiate a Flask application object
- Add code so that the Flask server is started with this file is executed directly by the Python interpreter
- Copy the `get_data()` method above into the application script
- Add a route (`/degrees`) which responds to the HTTP `GET` request and returns the complete list of data returned by `get_data()`

In a separate Terminal use `curl` to test out your new route. Does it work as expected?

Tip: Refer back to the Intro to Flask material if you need help remembering the boiler-plate code.

6.1.2 Responses in Flask

If you tried to return the list object directly in your route function definition, you got an error when you tried to request it with curl. Something like:

```
TypeError: The function did not return a valid response
```

Flask allows you three options for creating responses:

1) Return a string (`str`) object
2) Return a dictionary (`dict`) object
3) Return a tuple (`tuple`) object
4) Return a `flask.Response` object

Some notes:

- Option 1 is good for text or html such as when returning a web page or text file
- Option 2 is good for returning rich information in JSON-esque format
- Option 3 is good for returning a list of data using a special type of Python list - a `tuple` - which is ordered and unchangeable
- Option 4 gives you the most flexibility, as it allows you to customize the headers and other aspects of the response.

For our REST API, we will want to return JSON-formatted data. We will use a special Flask method to convert our list to JSON - `flask.jsonify`. (More on this later.)

Tip: Refer back to the Working with JSON material for a primer on the JSON format and relevant JSON-handling methods.

EXERCISE 2

Serialize the list returned by the `get_data()` method above into a JSON-formatted string using the Python `json` library. Verify that the type returned is a string.

Next, Deserialize the string returned in part a) by using the `json` library to decode it. Verify that the result equals the original list.

6.1.3 Returning JSON (and Other Kinds of Data)

You probably are thinking at this point we can fix our solution to **Exercise 1** by using the `json` library (which function?). Let's try that and see what happens:

EXERCISE 3

Update your code for Exercise 1 to use the `json` library to return a properly formatted JSON string.

Then, with your API server running in one window, open a Python3 interactive session in another window and:

- Make a `GET` request to your `/degrees` URL and capture the response in a variable, say `r`
- Verify that `r.status_code` is what you expect (what do you expect it to be?)
- Verify that `r.content` is what you expect
- Try to use `r.json()` to decode the response - does it work?
- Compare that with the response from the Bitbucket API to the URL `https://api.bitbucket.org/2.0/repositories`

The issue you may be encountering has to do with the `Content-Type` header being returned by the degrees API.

6.1.4 HTTP Content Type Headers

Requests and responses have `headers` which describe additional metadata about them. Headers are `key: value` pairs (much like dictionary entries). The `key` is called the header name and the `value` is the header value.

There are many pre-defined headers for common metadata such as specifying the size of the message (`Content-Length`), the domain the server is listening on (`Host`), and the type of content included in the message (`Content-Type`).

Media Type (or Mime Type)

The allowed values for the `Content-Type` header are the defined **media types** (formerly, **mime types**). The main thing you want to know about media types are that they:

- Consist of a type and subtype
- The most common types are application, text, audio, image, and multipart
- The most common values (type and subtype) are application/json, application/xml, text/html, audio/mpeg, image/png, and multipart/form-data

Content Types in Flask

The Flask library has the following built-in conventions you want to keep in mind:

- When returning a string as part of a route function in Flask, a `Content-Type` of text/html is returned
- To convert a Python object to a JSON-formatted string **and** set the content type properly, use the `flask.jsonify()` function.

For example, the following code will convert the list to a JSON string and return a content type of aplication/json:

```
return flask.jsonify(['a', 'b', 'c'])
```

EXERCISE 4

Use the `flask.jsonify()` method to update your code from Exercise 1. Then:

- Validate that your `/degrees` endpoint works as expected by using the `requests` library to make an API request and check that the `.json()` method works as expected on the response.
- Use the `.headers()` method on the response to verify the `Content-Type` is what you expect.

6.1.5 Query Parameters

The HTTP spec allows for parameters to be added to the URL in form of `key=value` pairs. Query parameters come after a `?` character and are separated by `&` characters; for example, the following request:

```
GET https://api.example.com/degrees?limit=3&offset=2
```

Passes two query parameters: `limit=3` and `offset=2`.

In REST architectures, query parameters are often used to allow clients to provide additional, optional arguments to the request.

Common uses of query parameters in RESTful APIs include:

- Pagination: specifying a specific page of results from a collection
- Search terms: filtering the objects within a collection by additional search attributes
- Other parameters that might apply to most if not all collections such as an ordering attribute (`ascending` vs `descending`)

Extracting Query Parameters in Flask

Flask makes the query parameters available on the `request.args` object, which is a "dictionary-like" object. To work with the query parameters supplied on a request, you must import the Flask request method (this is different from the Python3 `requests` library), and use an imbedded method to extract the passed query parameter into a variable:

```
from flask import Flask, request

@app.route('/degrees', methods=['GET'])
def degrees():
    start = request.args.get('start')
```

The `start` variable will be the value of the `start` parameter, if one is passed, or it will be `None` otherwise:

```
GET https://api.example.com/degrees?start=2
```

Note: `request.args.get()` will always return a `string`, regardless of the type of data being passed in.

EXERCISE 5

Add support for a `limit` parameter to the code you wrote for Exercise 4. The `limit` parameter should be optional. When passed with an integer value, the API should return no more than `limit` data points.

6.1.6 Additional Resources

- Flask JSON support
- Flask query parameter support

6.2 Containerizing Flask

As we have discussed previously, Docker containers are critical to packaging a (e.g. Flask) application along with all of its dependencies, isolating it from other applications and services, and deploying it consistently and reproducibly and platform-agnostically.

Here, we will walk through the process of containerizing Flask with Docker, then curling it as a containerized microservice. After going through this module, students should be able to:

- Assemble the different components needed for a containerized microservice into on directory
- Establish and document requirements (e.g. dependencies, Python packages) for the project
- Build and run in the background a containerized Flask microservice
- Map ports on the ISP server to ports inside a container, and curl the correct ports to return a response from the microservice

6.2.1 Organize Your App Directory

First, create a "web" directory, and change directories to it:

```
[isp02]$ mkdir web
[isp02]$ cd web
```

Then, create a new `app.py` (or copy an existing one) into this folder. It should have the following contents:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/', methods = ['GET'])
6 def hello_world():
7     return 'Hello, world!\n'
8
9 @app.route('/<name>', methods = ['GET'])
```

(continues on next page)

(continued from previous page)

```
def hello_name(name):
    return 'Hello, {}!\n'.format(name)

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

6.2.2 Establish Requirements

Python uses a specific file called `requirements.txt` to capture the required libraries and packages for a project. For our example here, create a file called `requirements.txt` and add the following line:

```
Flask==1.1.2
```

6.2.3 Build a Docker Image

As we saw in a previous section, we write up the recipe for our application install process in a Dockerfile. Create a file called `Dockerfile` for our Flask microservice and add the following lines:

```
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Here we see usage of the Docker `ENTRYPOINT` and `RUN` instructions, which essentially specify a default command (`python app.py`) that should be run when an instance of this image is instantiated.

Save the file and build the image with the following command:

```
[isp02]$ docker build -t <username>/flask-helloworld:latest .
```

6.2.4 Run a Docker Container

To create an instance of your image (a "container"), use the following command:

```
[isp02]$ docker run --name "give-your-container-a-name" -d -p <your port number>:5000
↪<username>/flask-helloworld:latest"
```

For example:

```
[isp02]$ docker run --name "charlies-helloworld-flask-app" -d -p 5050:5000 charlie/
↪flask-helloworld:latest"
```

The `-d` flag detaches your terminal from the running container - i.e. it runs the container in the background. The `-p` flag maps a port on the ISP server (5050, in the above case) to a port inside the container (5000, in the above case). In the above example, the Flask app was set up to use the default port inside the container (5000), and we can access that through our specified port on ISP (5050).

Check to see that things are up and running with:

```
[isp02]$ docker ps -a
```

The list should have a container with the name you gave it, an `UP` status, and the port mapping that you specified.

If the above is not found in the list of running containers, try to debug with the following:

```
[isp02]$ docker logs "your-container-name"
-or-
[isp02]$ docker logs "your-container-number"
```

6.2.5 Access Your Microservice

Now for the payoff - you can curl your REST API / Flask microservice by hitting the correct port on the ISP server. Following the example above, which was using port 5050:

```
[isp02]$ curl localhost:5050/
Hello, world!
[isp02]$ curl localhost:5050/Charlie
Hello, Charlie!
```

6.2.6 Clean Up

Finally, don't forget to stop your running container and remove it.

```
[isp02]$ docker ps -a | grep charlie
60be6788d73d   charlie/flask-helloworld:latest     "python app.py"   4 minutes ago
↪Up 4 minutes   0.0.0.0:5050->5000/tcp   charlies-helloworld-flask-app
[isp02]$ docker stop 60be6788d73d
60be6788d73d
[isp02]$ docker rm 60be6788d73d
60be6788d73d
```

EXERCISE

Note: This exercise will be reflected in Homework 03, part C.

Containerize your Dr. Moreau apps! Create a route that creates one random animal. Post a link to your route to Slack. Have another classmate hit your route, and build an animal.

CHAPTER

SEVEN

WEEK 8: INTRO TO DATABASES AND PERSISTENCE, CONTAINERIZING REDIS

In this eighth week of class, we will begin implementing a database using Redis to hold our data sets such that they persist beyond the lifetime of a container. We will also figure out how to query our data in Python by connecting our Flask API to our Redis database, and containerize both of our services.

7.1 Intro to Databases and Persistence

Application data that lives inside a container is ephemeral - it only persists for the lifetime of the container. We can use databases to extend the life of our application (or user) data, and even access it from outside the container.

After going through this module, students should be able to:

- Explain the differences between SQL and NoSQL databases
- Choose the appropriate type of database for a given application / data set
- Start and find the correct port for a Redis server
- Install and import the Redis Python library
- Add data to and retrieve data from a Redis database from a Python script

7.1.1 What's Our Motivation?

This week we work to extend our Flask App - which we will now refer to as our Flask API - to enable users to query and analyze our data sets.

Our basic approach to this will be:

1. Our dataset will be stored in our database
2. The user submits a request to a Flask endpoint which describes some sort of analysis they wish to perform
3. We will create functions to perform the analysis and retrieve the desired data from the database

---

Tip: For future lectures, think about the following: The analysis may take "a while" to execute, so we need to figure out how to (1) run the job in the background, (2) let the user know when the job has finished, and possibly (3) receive and handle multiple jobs at the same time. More on this coming soon!

7.1.2 Quick Intro to Databases

What is a database?

- A database is an organized collection of structured information, or data, typically stored electronically in a computer system

So why use one?

- Our data needs permanence and we want to be able to stop and start our Flask API without losing data
- We want multiple Python processes to be able to access the data at the same time

Why not use a file?

- It is not easy to make a file accessible to Python processes on different computers / VMs
- When multiple processes are reading from and writing to a file, race conditions can occur
- With files, our Flask API would have to implement all the structure in the data

NoSQL databases

- Yes, this implies a "Yes"SQL - or just SQL - database
- NoSQL databases do **NOT** use tables (data structured using rows and columns) connected through relations
- NoSQL databases store data in "collections" or "logical databases"
- Can allow for missing or different attributes on objects in the same collection
- Objects in one collection do not relate or link to objects in another collection
- The objects themselves could be JSON objects without a pre-defined schema

SQL vs NoSQL

- Both SQL and NoSQL databases have advantages and disadvantages
- The *primary* deciding factor should be the *shape* of the data
- Also consider how the data may change over time, and how important is the relationship between the data tables
- SQL "enforce" relationships between data types, including one-to-one, one-to-many, and many-to-many (important for some types of data; think hospitals or banks)
- In NoSQL, the relationship enforcement must be programmed into the application (think Twitter)
- SQL databases have challenges scaling to "large" quantities of data because of the ACID (Atomicity, Consistency, Isolation, Durability) guarantees they make
- NoSQL databases trade ACID guarantees for "eventual consistency" and greater scalability (i.e., a relational database would almost certainly not scale to "all tweets")

For the projects in this class, NoSQL is the way to go. We need a flexible data model as our 'animals' data structure keeps changing, we need something that is quick to get started, we need something that will allow our data to persist, and we need something to manage communication between our services.

7.1.3 Enter Redis

Redis is a very popular NoSQL database and "data structure store" with lots of advanced features including:

Key-value store

- The items stored in a Redis database are structured as `key:value` objects
- The primary requirement is that the `key` be unique across the database
- A single Redis server can support multiple databases, indexed by an integer
- The data itself can be stored as JSON

Notes about keys

- Keys are often strings, but they can be any "binary sequence"
- Long keys can lead to performance issues
- A format such as `<object_type>:<object_id>` is a good practice

Notes on values

- Values are typed; some of the primary types include:
  - Binary-safe strings
  - Lists (sorted collections of strings)
  - Sets (unsorted, unique collections of strings)
  - Hashes (maps of fields with associated values; both field and value are type `string`)
- There is no native "JSON" type; to store JSON, one can use an encoding and store the data as a binary-safe string, or one can use a hash and convert the object into and out of JSON
- The basic string type is a "binary-safe" string, meaning it must include an encoding
  - In Python terms, the string is stored and returned as type `bytes`
  - By default, the string will be encoded with UTF-8, but we can specify the encoding when storing the string
  - Since bytes are returned, it will be our responsibility to decode using the same encoding

Hash maps

- Hashes provide another way of storing dictionary-like data in Redis
- The values of the keys are type `string`

7.1.4 Running Redis

To use Redis on the class VM (ISP), we must have an instance of the Redis server running. For demonstration purposes, we will all share the same instance of Redis server on the same port (6379).

```
# start the Redis server on the command line:
[isp02]$ redis-server
3823:C 31 Mar 10:20:51.194 # Warning: no config file specified, using the default␣
↪config. In order to specify a config file use redis-server /path/to/redis.conf
3823:M 31 Mar 10:20:51.198 # You requested maxclients of 10000 requiring at least␣
↪10032 max file descriptors.
3823:M 31 Mar 10:20:51.198 # Server can't set maximum open files to 10032 because of␣
↪OS error: Operation not permitted.
3823:M 31 Mar 10:20:51.198 # Current maximum open files is 4096. maxclients has been␣
↪reduced to 4064 to compensate for low ulimit. If you need higher maxcli
↪increase 'ulimit -n'.
```

(continues on next page)

(continued from previous page)

```
3823:M 31 Mar 10:20:51.202 # Creating Server TCP listening socket *:6379: bind:␣
↪Address already in use

# already started! (remember, we are all logged in to the same VM)

# Ping the server to make sure it is up
[isp02]$ redis-cli ping
PONG
```

The Redis server is up and available. Although we could use the Redis CLI to interact with the server directly, in this class we will focus on the Redis Python library so we can interact with the server from our Python scripts.

Note: According to the log above, Redis is listening on the default port, **6379**.

First install the Redis Python library in your user account:

```
[isp02]$ pip3 install --user redis
```

Then open up an interactive Python interpreter to connect to the server:

```
[isp02]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import redis
>>>
>>> rd=redis.StrictRedis(host='127.0.0.1', port=6379, db=0)
>>>
>>> type(rd)
<class 'redis.client.Redis'>
```

You've just created a client connection to the Redis server called `rd`. This class contains methods for adding, modifying, deleting, and analyzing data in the database instance, among other things.

Some quick notes:

- We are using the IP of the gateway (`127.0.0.1`) on our localhost and the default Redis port (`6379`).
- Redis organizes collections into "databases" identified by an integer index. Here, we are specifying `db=0`; if that database does not exist it will be created for us.

7.1.5 Working with Redis

We can create new entries in the dabase using the `.set()` method. Remember, entries in a Redis database take the form of a key:value pair. For example:

```
>>> rd.set('my_key', 'my_value')
True
```

This operation saved a key in the Redis server (`db=0`) called `my_key` and with value `my_value`. Note the method returned True, indicating that the request was successful.

We can retrieve it using the `.get()` method:

```
>>> rd.get('my_key')
b'my_value'
```

Note that `b'my_value'` was returned; in particular, Redis returned binary data (i.e., type `bytes`). The string was encoded for us (in this case, using Unicode). We could have been explicit and set the encoding ourselves. The `bytes` class has a `.decode()` method that can convert this back to a normal string, e.g.:

```
>>> rd.get('my_key')
b'my_value'
>>> type(rd.get('my_key'))
<class 'bytes'>
>>>
>>> rd.get('my_key').decode('utf-8')
'my_value'
>>> type( rd.get('my_key').decode('utf-8') )
<class 'str'>
```

Exercise 1

With this knowledge, write a Python program that:

- Uses a loop to create 10 random numbers and chooses a random heads
- Store the random number as a `key` and the random head as the `value`

7.1.6 Redis and JSON

A lot of the information we exchange comes in JSON or Python dictionary format. To store pure JSON as a binary-safe string `value` in a Redis database, we need to be sure to dump it as a string (`json.dumps()`):

```
>>> import json
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> rd.set('k1', json.dumps(d))
True
```

Retrieve the data again and get it back into JSON / Python dictionary format using the `json.loads()` method:

```
>>> rd.get('k1')
b'{"a": 1, "b": 2, "c": 3}'
>>> type(rd.get('k1'))
<class 'bytes'>
>>>
>>> json.loads(rd.get('k1'))
{'a': 1, 'b': 2, 'c': 3}
>>> type(json.loads(rd.get('k1')))
<class 'dict'>
```

Note: In some versions of Python, you may need to specify the encoding as we did earlier, e.g.:

```
>>> json.loads(rd.get('k1').decode('utf-8'))
{'a': 1, 'b': 2, 'c': 3}
```

Hashes

Hashes provide another way of storing dictionary-like data in Redis.

- Hashes are useful when different fields are encoded in different ways; for example, a mix of binary and unicode data.
- Each field in a hash can be treated with a separate decoding scheme, or not decoded at all.
- Use `hset()` to set a single field value in a hash; use `hmset()` to set multiple fields at once.
- Similarly, use `hget()` to get a single field within a hash, use `hmget()` to get multiple fields, or use `hgetall()` to get all of the fields.

```
>>> rd.hmset('k2', {'name': 'Charlie', 'email': 'charlie@tacc.utexas.edu'})
>>> rd.hgetall('k2')
{b'name': b'Charlie', b'email': b'charlie@tacc.utexas.edu'}

>>> rd.hset('k2', 'name', 'Charlie Dey')
>>> rd.hgetall('k2')
{b'name': b'Charlie Dey', b'email': b'charlie@tacc.utexas.edu'}

>>> rd.hget('k2', 'name')
b'Charlie Dey'

>>> rd.hmget('k2', 'name', 'email')
[b'Charlie Dey', b'charlie@tacc.utexas.edu']
```

Tip: You can use `rd.keys()` to return all keys from a database, and `rd.hkeys(key)` to return the list of keys within hash '`key`', e.g.:

```
>>> rd.hkeys('k2')
[b'name', b'email']
```

Exercise 2

Modify your animal producer - your app that creates your animals - to write out five animals to the Redis database. Use a random number as the key and a hash as your value.

Exercise 3

Create another animal consumer - your app that reads in the animals - to read in five random animals from the database using a random key.

Exercise 4

Modify your animal consumer to read in all the animals with a specific type of head.

> **Warning:** What happens when a key is not found? How can we adjust our code for this?

7.1.7 Additional Resources

- Redis Docs
- Redis Python Library

7.2 Containerizing Redis

Up to this point, we have been interacting with a shared Redis database instance directly on the class ISP server. Next, we will each containerize an instance of Redis, figure out how to interact with it by forwarding the port, and connect it to our Flask app.

After going through this module, students should be able to:

- Start a Redis container, connecting the appropriate inside port to a port on ISP
- Connect to the container from within a Python script
- Mount a volume inside the container for data persistence

7.2.1 Start a Redis Container

Docker Hub has a wealth of official, public images. It is a good idea to pull the existing Redis image rather than build it ourself, because it has all of the functionality we need as a base image.

Pull the official Redis image for version 5.0.0:

```
[isp02]$ docker pull redis:5.0.0
5.0.0: Pulling from library/redis
Digest: sha256:481678b4b5ea1cb4e8d38ed6677b2da9b9e057cf7e1b6c988ba96651c6f6eff3
Status: Image is up to date for redis:5.0.0
docker.io/library/redis:5.0.0
```

Start the Redis server in a container:

```
[isp02]$ docker run -p <your-redis-port>:6379 redis:5.0.0
1:C 31 Mar 2021 16:48:11.939 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
...
1:M 31 Mar 2021 16:48:11.972 * Ready to accept connections
```

The above command will start the Redis container in the foreground which can be helpful for debugging, seeing logs, etc. However, you will have the need to start it in the background (detached or daemon mode).

```
[isp02]$ docker run -d -p <your-redis-port>:6379 redis:5.0.0
3a28cb265d5e09747c64a87f904f8184bd8105270b8a765e1e82f0fe0db82a9e
```

7.2.2 Connect to the Container from Python

Only one small change is needed in our Python scripts to connect to this containerized version of Redis. Since Docker is smart about port forwarding to the localhost, you simply need to change the port you connect to when setting up the Redis client.

```
>>> import redis
>>> rd=redis.StrictRedis(host='127.0.0.1', port=<your-redis-port>, db=0)
```

Exercise 1

Revisit the Exercises from the previous module Modify them to hit your Redis container instead.

Exercise 2

Kill your Redis container and restart it. Is your data still there?

Tip: Use `docker ps` to find the container ID, then `docker rm -f <containerID>`

Exercise 3

Alright, so when we took down our Redis container, and then brought it back… we lost our data? What can we do about this?

Mount a **volume** in your running container:

```
$ docker run -d -p <your port>:6379 -v <data-dir>:/data redis:5.0.0
```

7.2.3 Additional Resources

- Redis Image on Docker Hub

CHAPTER

EIGHT

WEEK 9: CONTAINER ORCHESTRATION WITH KUBERNETES

In Week 9, we begin our study of container orchestration and the Kubernetes ("k8s") system. We have already built a small HTTP application in the REST architecture using the flask framework. This HTTP application makes use of a database to persist state. In the coming weeks, we will add more components to our application, and this is very typical of a modern distributed system. As the number of components grows, the work required to deploy and maintain this system increases. Container orchestration systems such as k8s aid us in this deployment and management effort by allowing us to run our applications across a cluster of machines and use APIs to make changes to the application deployment over time.

At the end of the next two weeks you will:

- Understand container orchestration and the basic Kubernetes architecture.
- Understand fundamental Kubernetes abstractions, including: `pod`, `deployment`, `persistent volume`, and `service`.
- Write a basic script to deploy your flask application to a Kubernetes cluster in your own private namespace.

8.1 Docker Compose, Revisited

Let's do a bit of rehash: docker-compose is a tool for defining and running multi-container Docker applications. With docker-compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Here, we will revisit our containerized Flask and Redis services, fire them up with docker-compose, and then have our two services talk to each other.

When working with multiple containers, it can be difficult to manage the starting configuration along with the variables and links. Docker-compose is one orchestration tool for defining and running multi-container Docker applications.

8.1.1 Why Docker-Compose?

- Orchestration!
- Launch multiple containers with complex configurations at once
- Define the structure of your app, not the commands needed to run it!

8.1.2 Using Docker-Compose

Using docker-compose is a three-step process:

- Define images with Dockerfiles
- Define the services in a docker-compose.yml as containers with all of your options (e.g. image, port mapping, links, etc.)
- Run `docker-compose up` and it starts and runs your entire app

Three step process to use ... a bit more to actually build.

8.1.3 Orchestrating Redis

The first thing to do is create a new Docker build context for our app - this is the collection of files and folders that goes in to the image(s) we build. Create a folder called `redis-docker` and directories called `config` and `data` within:

```
[isp02]$ mkdir redis-docker/
[isp02]$ mkdir redis-docker/config
[isp02]$ mkdir redis-docker/data
```

Next copy this redis config file into your config directory. We are going to put this into our Redis container, and it will allow us to customize the behavior of our database server, if desired.

```
[isp02]$ cd redis-docker
[isp02]$ wget -O config/redis.conf https://raw.githubusercontent.com/TACC/coe-332-
↪sp21/main/docs/week09/redis.conf
[isp02]$ ls config/
redis.conf
```

Now in your top directory, create a new file called `docker-compose.yml`. Populate the file with the following contents, being careful to preserve indentation, and replacing the username / port placeholders with your own:

```
---
version: '3'
services:
    redis:
        image: redis:latest
        container_name: <your username>-redis
        ports:
            - <your redis port>:6379
        volumes:
            - ./config/redis.conf:/redis.conf
        command: [ "redis-server", "/redis.conf" ]
```

8.1.4 Start the Redis Service

Bring up your Redis container with the following command:

```
[isp02]$ docker-compose -p <your username> up -d
```

Take note of the following options:

- `docker-compose up` looks for docker-compose.yml and starts the services described within
- `-p <your username>` gives the project a unique name, which will help avoid collisions with other student's containers
- `-d` puts it in daemon mode (runs in the background)

Check to see if your Redis database is up and the port is forwarding as you expect with the following:

```
[isp02]$ docker ps
CONTAINER ID   IMAGE            COMMAND                  CREATED          STATUS
↪     PORTS                    NAMES
aa1b2b6908a9   redis:5.0.0      "docker-entrypoint.s..."   58 seconds ago   Up 55
↪seconds    0.0.0.0:6080->6379/tcp   charlie-redis

[isp02]$ docker logs aa1b2b6908a9
1:C 31 Mar 2021 20:14:45.615 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 31 Mar 2021 20:14:45.615 # Redis version=6.2.1, bits=64, commit=00000000,
↪modified=0, pid=1, just started
1:C 31 Mar 2021 20:14:45.615 # Configuration loaded
1:M 31 Mar 2021 20:14:45.618 * monotonic clock: POSIX clock_gettime
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 6.2.1 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6080
      |    `-._   `._    /     _.-'    |     PID: 1
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

1:M 31 Mar 2021 20:14:45.623 # WARNING: The TCP backlog setting of 511 cannot be
↪enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
1:M 31 Mar 2021 20:14:45.623 # Server initialized
1:M 31 Mar 2021 20:14:45.623 # WARNING overcommit_memory is set to 0! Background save
↪may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1
↪' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_
↪memory=1' for this to take effect.
1:M 31 Mar 2021 20:14:45.625 * Ready to accept connections
```

Boom! We have Redis running!

But Charlie! *docker-compose is about defining and running multi-container Docker applications!*

8.1.5 Add the Flask Service

First let's take down the existing service:

```
[isp02]$ docker-compose -p <your username> down
Stopping charlie-redis ... done
Removing charlie-redis ... done
Removing network charlie_default
```

Note: It is assumed you are still in the same directory as your docker-compose.yml file, if not otherwise specified.

Next, add the following new lines to your docker-compose.yml file:

```
---
version: '3'
services:
    redis:
        image: redis:latest
        container_name: <your username>-redis
        ports:
            - <your redis port>:6379
        volumes:
            - ./config/redis.conf:/redis.conf
        command: [ "redis-server", "/redis.conf" ]
    web:
       build: .
       container_name: <your-username>-web
       ports:
          - <your flask port>:5000
       volumes:
          - ./data/data_file.json:/datafile.json
```

With these lines, you are adding a new service called 'web'. Take care to replace the placeholders with your assigned Redis port and Flask port numbers. Note Redis and Flask use default ports 6379 and 5000, respectively, inside the containers unless otherwise specified.

Also new to this service, we are using the `build` key to build a new Docker image based on the files / Dockerfile in this (`.`) directory. We need to pull in our web assets (wherever they are located - it may be different for each person) and Dockerfile from our previous exercises to this current directory.

```
[isp02]$ mkdir web
[isp02]$ cp ~/coe-332/web1/app.py ./web/
[isp02]$ cp ~/coe-332/web1/requirements.txt ./
[isp02]$ cp ~/coe-332/web1/data_file.json ./data/
[isp02]$ cp ~/coe-332/web1/Dockerfile ./

# Now your directory structure should look like:
[isp02]$ tree .
.
├── config
│   └── redis.conf
├── data
│   └── data_file.json
├── docker-compose.yml
├── Dockerfile
└── web
```

(continues on next page)

(continued from previous page)

```
    ├── app.py
    └── requirements.txt
```

This time when you start services, two containers will be created, one of which is built from the current directory.

```
[isp02]$ docker-compose -p charlie up -d
Creating network "charlie_default" with the default driver
Creating charlie-redis ... done
Creating charlie-web   ... done
```

8.1.6 Modify Python Redis Client

When you do `docker-compose up`, behind the scenes Docker creates a custom bridge network for each of your services to talk to one another. They can reach each other using the name of the service as the 'host', e.g.:

```
>>> rd = redis.StrictRedis(host='redis', port=6379, db=0)
```

Exercise

Connect your Flask container and your Redis container together using docker-compose, and curl the various endpoints to make sure it works.

Note: Be sure to change your Redis connection in your Flask App!

8.2 Orchestration Overview

A typical distributed system is comprised of multiple components. You have already developed a simple HTTP application that includes a Python program (using the flask library) and a database. In the subsequent weeks, we will add additional components.

Container orchestration involves deploying and managing containerized applications on one or more computers. There are a number of challenges involved in deploying and managing a distributed application, including:

- Container execution and lifecycle management – To run our application, we must not only start the containers initially but also manage the entire lifecycle of the container, including handling situations where containers are stopped and/or crash. We need to ensure the correct container image is used each time.
- Configuration – Most nontrivial applications involve some configuration. We must be able to provide configuration to our application's components.
- Networking – The components in our distributed application will need to communicate with each other, and that communication will take place over a network.
- Storage – Some components, such as databases, will require access to persistent storage to save data to disc so that they can be restarted without information loss.

The above list is just an initial set of concerns when deploying distributed, containerized applications. As the size and number of components grows, some systems may encounter additional challenges, such as:

- Scaling – We may need to start up additional containers for one or more components to handle spikes in load on the system, and shut down these additional containers to save resources when the usage spike subsides.

- CPU and Memory management – The computers we run on have a fixed amount of CPU and memory, and in some cases, it can be important to ensure that no one container uses too many resources, or to ensure that
- Software version updates – How do we go from version 1 to version 2 of our software? We may have to update several components (or all of them) at once. What if there are multiple containers running for a given component? As we are performing the upgrade, is the system offline or can users still use it?

8.2.1 Orchestration Systems

Orchestration systems are built to help with one ore more of the above challenges. Below we briefly cover some of the more popular, open-source container orchestration systems. This is by no means an exhaustive list.

Docker Compose

You have already seen a basic container orchestration system – Docker Compose. The Docker Compose system allows users to describe their application in a `docker-compose.yml` file, including the `services`, `networks`, `volumes` and other aspects of the deployment. Docker Compose:

- Runs Docker containers on a single computer, the machine where docker-compose is installed and run.
- Utilizes the Docker daemon to start and stop containers defined in the `docker-compose.yml` file.
- Also capable of creating volumes, networks, port bindings, and other objects available in the Docker API.

Docker compose is a great utility for single-machine deployments and, in particular, is a great system for "local development environments" where a developer has code running on her or his laptop.

Docker Swarm

Docker Swarm provides a container orchestration system to deploy applications across multiple machines running the Docker daemon. Docker Swarm works by creating a cluster (also known as a "swarm") of computers and coordinating the starting and stopping of containers across the cluster. Docker Swarm:

- Runs Docker containers across a cluster of machines (a "swarm"), each running Docker.
- Coordinates container execution across the cluster.
- Similar API to Docker Compose: capable of creating networks spanning multiple computers as well as port-bindings, volumes, etc.

Mesos

Apache Mesos is a general-purpose cluster management system for deploying both containerized and non-containerized applications across multiple computers. Mesos by itself is quite low-level and requires the use of *frameworks* to deploy actual applications. For example, Marathon is a popular Mesos framework for deploying containerized applications, while the Mesos Hydra framework can be used for deploying MPI-powered applications, such as those used in traditional HPC applications.

Kubernetes

Kubernetes (often abbreviated as "k8s") is a container orchestration system supporting Docker as well other container runtimes that conform to the Container Runtime Interface (CRI) such as containerd and cri-o. While Kubernetes focuses entirely on containerized applications (unlike Mesos) and is not as similar to Docker Compose as Docker Swarm is, it provides a number of powerful features for modern, distributed systems management. Additionally, Kubernetes is available as a service on a large number of commercial cloud providers, including Amazon, Digital Ocean, Google, IBM, Microsoft, and many more, and TACC provides multiple Kubernetes clusters in support of various research projects.

- Supports running containerized applications across a cluster of machines for container runtimes conforming to the Container Runtime Interface (CRI), including Docker.
- Provides powerful features for managing distributed applications.
- Available as a service from TACC and a number of commercial cloud providers.

8.2.2 Additional Resources

- Docker Compose Reference
- Docker Swarm
- Apache Mesos Documentation
- Marathon
- Mesos Hydra
- Kubernetes Documentation

8.3 Kubernetes - Overview and Introduction to Pods

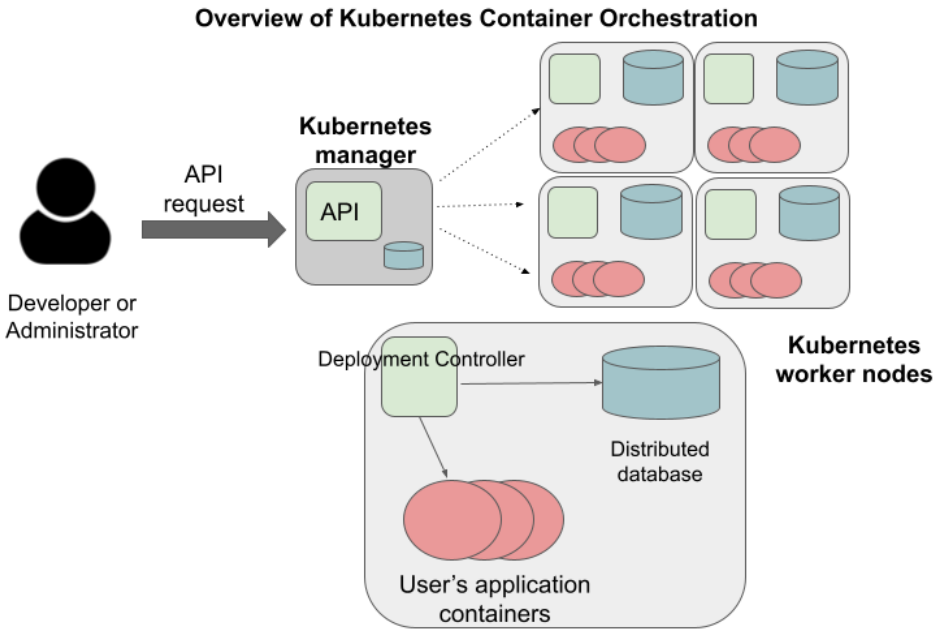
In this section we give an overview of the Kubernetes system and introduce the first major Kubernetes abstraction, the Pod.

8.3.1 Kubernetes Overview

Kubernetes (k8s) is itself a distributed system of software components that run a cluster of one or more machines (physical computers or virtual machines). Each machine in a k8s cluster is either a "manager" or a "worker" node.

Users communicate with k8s by making requests to its API. The following steps outline how Kubernetes works at a high level:

1) Requests to k8s API describe the user's *desired state* on the cluster; for example, the desire that 3 containers of a certain image are running.
2) The k8s API schedules new containers to run on one or more worker nodes.
3) After the containers are started, the Kubernetes deployment controller, installed on each worker node, monitors the containers on the node.
4) The k8s components, including the API and the deployment controllers, maintain both the *desired state* and the *actual state* in a distributed database. The components continuously coordinate together to make the actual state converge to the desired state.



Note: It is important to note that most of the time, the k8s API as well as the worker nodes are running on separate machines from the machine we use to interact with k8s (i.e., make API requests to it). The machine we use to interact with k8s only needs to have the k8s client tools installed, and in fact, as the k9s API is available over HTTP, we don't strictly speaking require the tools – we could use curl or some other http client – but the tools make interacting much easier.

Connecting to the TACC Kubernetes Instance

In this class, we will use TACC's FreeTail Kubernetes cluster for deploying our applications. To simplify the process of using FreeTail, we have enabled connectivity to it from the ips02 host. Therefore, any time you want to work with k8s, simply SSH to isp02 with your TACC username as you have throughout the semester:

```
$ ssh <tacc_username>@isp02.tacc.utexas.edu
```

You will be prompted for your TACC username and password, just as you are when connecting to isp02.

First Commands with k8s

We will use the Kubernetes Command Line Interface (CLI) referred to as "kubectl" (pronounced "Kube control") to make requests to the Kubernetes API. We could use any HTTP client, including a command-line client such as curl, but `kubectl` simplifies the process of formatting requests.

The `kubectl` software should already be installed and configured to use the Freetail K8s cluster. Let's verify that is the case by running the following:

```
$ kubectl version -o yaml
```

You should see output similar to the following:

```
clientVersion:
  buildDate: "2021-01-13T13:28:09Z"
  compiler: gc
  gitCommit: faecb196815e248d3ecfb03c680a4507229c2a56
  gitTreeState: clean
  gitVersion: v1.20.2
  goVersion: go1.15.5
  major: "1"
  minor: "20"
  platform: linux/amd64
serverVersion:
  buildDate: "2020-11-11T13:09:17Z"
  compiler: gc
  gitCommit: d360454c9bcd1634cf4cc52d1867af5491dc9c5f
  gitTreeState: clean
  gitVersion: v1.19.4
  goVersion: go1.15.2
  major: "1"
  minor: "19"
  platform: linux/amd64
```

This command made an API request to the TACC Freetail k8s cluster and returned information about the version of k8s running there (under `serverVersion`) as well as the version of the `kubectl` that we are running (under `clientVersion`).

Note: The output of the `kubectl` command was yaml because we used the `-o yaml` flag. We could have asked for the output to be formatted in json with `-o json`. The `-o` flag is widely available on `kubectl` commands.

8.3.2 Introduction to Pods

Pods are a fundamental abstraction within Kubernetes and are the most basic unit of computing that can be deployed onto the cluster. A pod can be thought of as generalizing the notion of a container: a pod contains one or more containers that are tightly coupled and need to be scheduled together, on the same computer, with access to a shared file system and a shared network address.

Note: By far, the majority pods you will meet in the wild, including the ones used in this course, will only include one container. A pod with multiple containers can be thought of as an "advanced" use case.

8.3.3 Hello, Kubernetes

To begin, we will define a pod with one container. As we will do with all the resources we want to create in k8s, we will describe our pod in a yaml file.

Create a file called `pod-basic.yml`, open it up in an editor and paste the following code in:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
```

(continues on next page)

(continued from previous page)

```
  containers:
    - name: hello
      image: ubuntu:18.04
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let's break this down. The top four attributes are common to all k8s resource descriptions:

- `apiVersion` – describes what version of the k8s API we are working in. We are using `v1`.
- `kind` – tells k8s what kind of resource we are describing, in this case a `Pod`.
- `metadata` – in general, this is additional information about the resource we are describing that doesn't pertain to its operation. Here, we are giving our pod a `name`, `hello`.
- `spec` – This is where the actual description of the resource begins. The contents of this stanza vary depending on the `kind` of resource you are creating. We go into more details on this in the next section.

Warning: Only one Kubernetes object of a specific `kind` can have a given `name` at a time. If you define a second pod with the same name you will overwrite the first pod. This is true of all the different types of k8s objects we will be creating.

8.3.4 The Pod Spec

In k8s, you describe resources you want to create or update using a `spec`. The required and optional parameters available depend on the `kind` of resource you are describing.

The pod spec we defined looked like this:

```
spec:
  containers:
    - name: hello
      image: ubuntu:18.04
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

There is just one stanza, the `containers` stanza, which is a list of containers (recall that pods can contain multiple containers). Here we are defining just one container. For each container, we provide:

- `name` (optional) – this is the name of the container, similar to the name attribute in Docker.
- `image` (required) – the image we want to use for the container, just like with Docker.
- `command` (optional) – the command we want to run in the container. Here we are running a short BASH script.

8.3.5 Creating the Pod In K8s

We are now ready to create our pod in k8s. To do so, we use the `kubectl apply` command. In general, when you have a description of a resource that you want to create or update in k8s, the `kubectl apply` command can be used.

In this case, our description is contained in a file, so we use the `-f` flag. Try this now:

```
$ kubectl apply -f pod-basic.yml
```

If all went well and k8s accepted your request, you should see an output like this:

```
pod/hello created
```

In practice, we won't be creating many `Pod` resources directly – we'll be creating other resources, such as `deployments` that are made up of pods – but it is important to understand pods and to be able to work with pods using `kubectl` for debugging and other management tasks.

Note: The pod we just created is running on the FreeTail k8s cluster, NOT on isp02. You will not be able to find it using commands like docker ps, etc.

8.3.6 Working With Pods

We can use additional `kubectl` commands to get information about the pods we run on k8s.

Listing Pods

For example, we can list the pods on the cluster with `kubectl get <object_type>` – in this case, the object type is "pods":

```
$ kubectl get pods

  NAME                              READY   STATUS    RESTARTS   AGE
  ds-staging-6df657fbd-tbht5        1/1     Running   0          34d
  elasticsearch-748f666f4f-svf2h    1/1     Running   0          76d
  hello                             1/1     Running   0          3s
  kibana-f5b79569f-r4pl6            1/1     Running   0          78d
  sidecartest-5454b7d49b-q8fvw      3/3     Running   472        78d
```

The output is fairly self-explanatory. We see a line for every pod which includes its name, status, the number of times it has been restarted and its age. Our `hello` pod is listed above, with an age of `3s` because we just started it but it is already RUNNING. Several additional pods are listed in my output above due to prior work sessions.

A Word on Authentication and Namespaces

With all the students running their own pods on the same k8s cluster, you might be wondering why you only see your pod or why you don't see my pods? The reason is that when you make an API request to k8s, you tell the API who you are and what *namespace* you want to make the request in. Namespaces in k8s are logically isolated views or partitions of the k8s objects. Your `kubectl` client is configured to make requests in a namespace that is private to you; we set these namespaces up for COE 332.

Getting and Describing Pods

We can pass the pod name to the `get` command – i.e., `kubectl get pods <pod_name>` – to just get information on a single pod

```
$ kubectl get pods hello
  NAME    READY   STATUS    RESTARTS   AGE
  hello   1/1     Running   0          3m1s
```

The `-o wide` flag can be used to get more information:

```
$ kubectl get pods hello -o wide
  NAME    READY   STATUS    RESTARTS   AGE    IP             NODE    NOMINATED NODE  ␣
↪READINESS GATES
  hello   1/1     Running   0          3m1s   10.244.5.28    c04     <none>
↪<none>
```

Finally, the `kubectl describe <resource_type> <resource_name>` command gives additional information, including the k8s events at the bottom. While we won't go into the details now, this information can be helpful when troubleshooting a pod that has failed:

```
$ kubectl describe pods hello
  Name:         hello
  Namespace:    designsafe-jupyter-stage
  Priority:     0
  Node:         c04/172.16.120.11
  Start Time:   Fri, 26 Feb 2021 10:12:43 -0600
  Labels:       <none>
  Annotations:  <none>
  Status:       Running
  IP:           10.244.5.28
  IPs:
    IP:  10.244.5.28
  Containers:
    hello:
      Container ID:  containerd://
↪b0e2d0eb8dc7717567886c99cfb30b9245c99f2b2f3a6610d5d6fe24fe8866b8
      Image:         ubuntu:18.04
      Image ID:      docker.io/library/ubuntu:18.
↪04@sha256:c6b45a95f932202dbb27c31333c4789f45184a744060f6e569cc9d2bf1b9ad6f
      Port:          <none>
      Host Port:     <none>
      Command:
        sh
        -c
        echo "Hello, Kubernetes!" && sleep 3600
      State:          Running
        Started:      Mon, 01 Mar 2021 11:14:38 -0600
      Last State:     Terminated
        Reason:       Completed
        Exit Code:    0
        Started:      Mon, 01 Mar 2021 10:14:37 -0600
        Finished:     Mon, 01 Mar 2021 11:14:37 -0600
      Ready:          True
      Restart Count:  73
      Environment:    <none>
      Mounts:
        /var/run/secrets/kubernetes.io/serviceaccount from default-token-xpg9m (ro)
  Conditions:
    Type              Status
    Initialized       True
    Ready             True
    ContainersReady   True
    PodScheduled      True
  Volumes:
    default-token-xpg9m:
      Type:        Secret (a volume populated by a Secret)
      SecretName:  default-token-xpg9m
```

(continues on next page)

(continued from previous page)

```
    Optional:    false
  QoS Class:       BestEffort
  Node-Selectors:  <none>
  Tolerations:     node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                   node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
  Events:
    Type    Reason   Age                    From     Message
    ----    ------   ----                   ----     -------
    Normal  Pulling  9m32s (x74 over 3d1h)  kubelet  Pulling image "ubuntu:18.04"
    Normal  Created  9m31s (x74 over 3d1h)  kubelet  Created container hello
    Normal  Started  9m31s (x74 over 3d1h)  kubelet  Started container hello
    Normal  Pulled   9m31s                  kubelet  Successfully pulled image
↪"ubuntu:18.04" in 601.12516ms
```

Getting Pod Logs

Finally, we can use `kubectl logs <pod_name>` command to get the logs associated with a pod:

```
$ kubectl logs hello
  Hello, Kubernetes!
```

Note that the `logs` command does not include the resource name ("pods") because it only can be applied to pods. The `logs` command in k8s is equivalent to that in Docker; it returns the standard output (stdout) of the container.

Using Labels

In the pod above we used the `metadata` stanza to give our pod a name. We can use `labels` to add additional metadata to a pod. A label in k8s is nothing more than a `name: value` pair that users create to organize objects and add meaningful to the user. We can choose any value for `name` and `value` that we wish but they must be strings. If you want to use a number like "10" for a label name or value, be sure to enclose it in quotes (i.e., `10`).

You can think of these `name:value` pairs as variables and values. So for example, you might a label called `shape` with values `circle`, `triangle`, `square`, etc. Multiple pods can have the same `name:value` label.

Let's use the pod definition above to create a new pod with a label.

Create a file called `pod-label.yml`, open it up in an editor and paste the following code in:

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: hello-label
  labels:
    version: "1.0"
spec:
  containers:
    - name: hello
      image: ubuntu:18.04
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let's create this pod using `kubectl apply`:

```
$ kubectl apply -f pod-label.yml
pod/hello-label created
```

Now when we list our pods, we should see it

```
$ kubectl get pods
  NAME                              READY   STATUS    RESTARTS   AGE
  ds-staging-6df657fbd-tbht5        1/1     Running   0          34d
  elasticsearch-748f666f4f-svf2h    1/1     Running   0          76d
  hello                             1/1     Running   0          4m
  hello-label                       1/1     Running   0          4s
  kibana-f5b79569f-r4pl6            1/1     Running   0          78d
  sidecartest-5454b7d49b-q8fvw      3/3     Running   472        78d
```

Filtering By Labels With Selectors

Labels are useful because we can use `selectors` to filter our results for a given label name and value. To specify a label name and value, use the following syntax: `--selector "<label_name>=<label_value>"`.

For instance, we can search for pods with the version 1.0 label like so:

```
$ kubectl get pods  --selector "version=1.0"
  NAME          READY   STATUS    RESTARTS   AGE
  hello-label   1/1     Running   0          4m58s
```

We can also just use the label name to filter with the syntax `--selector "<label_name>"`. This will find any pods with the label `<label_name>`, regardless of the value.

8.3.7 Additional Resources

- k8s Pod Reference

CHAPTER

NINE

WEEK 10: CONTAINER ORCHESTRATION WITH KUBERNETES, CONTINUED

In this week, we complete our treatment of the Kubernetes orchestration system by introducing two new resource types: Deployments and Services. We will be using deployments to describe our applications and we will use services to expose our applications to the network.

9.1 Deployments

Deployments are an abstraction and resource type in Kubernetes that can be used to represent long-running application components, such as databases, REST APIs, web servers, or asynchronous worker programs. The key idea with deployments is that they should *always be running*.

Imagine a program that runs a web server for a blog site. The blog website should always be available, 24 hours a day, 7 days a week. If the blog web server program crashes, it would ideally be restarted immediately so that the blog site becomes available again. This is the main idea behind deployments.

Deployments are defined with a pod definition and a replication strategy, such as, "run 3 instances of this pod across the cluster" or "run an instance of this pod on every worker node in the k8s cluster."

For this class, we will define deployments for our flask application and its associated components, as deployments come with a number of advantages over defining "raw" pods. Deployments:

- Can be used to run multiple instances of a pod, to allow for more computing to meet demands put on a system.
- Are actively monitored by k8s for health – if a pod in a deployment crashes or is otherwise deemed unhealthy, k8s will try to start a new one automatically.

9.1.1 Creating a Basic Deployment

We will use yaml to describe a deployment just like we did in the case of pods. Copy and paste the following into a file called `deployment-basic.yml`

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 1
  selector:
```

(continues on next page)

(continued from previous page)

```
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Let's break this down. Recall that the top four attributes are common to all k8s resource descriptions, however it is worth noting:

- `apiVersion` – We need to use version `apps/v1` here. In k8s, different functionalities are packaged into different APIs. Deployments are part of the `apps/v1` API, so we must specify that here.
- `metadata` – The `metadata.name` gives our deployment object a name. This part is similar to when we defined pods. However, the `labels` concept is new. k8s uses labels to allow objects to refer to other objects in a decoupled way. A label in k8s is nothing more than a `name: value` pair that users create to organize objects and add information meaningful to the user. In this case, `app` is the name and `hello-app` is the value. Conceptually, you can think of label names like variables and labels values as the value for the variable. In some other deployment, we may choose to use label `app: profiles` to indicate that the deployment is for the "profiles" app.

Let's look at the `spec` stanza for the deployment above.

- `replicas` – Defines how many pods we want running at a time for this deployment, in this case, we are asking that just 1 pod be running at a time.
- `selector` – This is how we tell k8s where to find the pods to manage for the deployment. Note we are using labels again here, the `app: hello-app` label in particular.
- `template` – Deployments match one or more pod descriptions defined in the template. Note that in the `metadata` of the template, we provide the same label (`app: hello-app`) as we did in the `matchLabels` stanza of the `selector`. This tells k8s that this spec is part of the deployment.
- `template.spec` – This is a pod spec, just like we worked with last time.

Note: If the labels, selectors and matchLables seems confusing and complicated, that's understandable. These semantics allow for complex deployments that dynamically match different pods, but for the deployments in this class, you will not need this extra complexity. As long as you ensure the label in the `template` is the same as the label in the `selector.matchLables` your deployments will work. It's worth pointing out that the first use of the `app: hello-app` label for the deployment itself (lines 5 and 6 of the yaml) could be removed without impacting the end result.

We create a deployment in k8s using the `apply` command, just like when creating a pod:

```
$ kubectl apply -f deployment-basic.yml
```

If all went well, k8s response should look like:

```
deployment.apps/hello-deployment created
```

We can list deployments, just like we listed pods:

```
$ kubectl get deployments
  NAME               READY    UP-TO-DATE    AVAILABLE    AGE
  hello-deployment   1/1      1             1            1m
```

We can also list pods, and here we see that k8s has created a pod for our deployment for us:

```
$ kubectl get pods
  NAME                                READY    STATUS    RESTARTS   AGE
  hello                               1/1      Running   0          29m
  hello-deployment-9794b4889-kms7p    1/1      Running   0          1m
```

Note that we see our "hello" pod from earlier as well as the pod "hello-deployment-9794b4889-kms7p" that k8s created for our deployment. We can use all the kubectl commands associated with pods, including listing, describing and getting the logs. In particular, the logs for our "hello-deployment-9794b4889-kms7p" pod prints the same "Hello, Kubernetes!" message, just as was the case with our first pod.

9.1.2 Deleting Pods

However, there is a fundamental difference between the "hello" pod we created before and our "hello" deployment which we have alluded to. This difference can be seen when we delete pods.

To delete a pod, we use the `kubectl delete pods <pod_name>` command. Let's first delete our hello deployment pod:

```
$ kubectl delete pods hello-deployment-9794b4889-kms7p
```

It might take a little while for the response to come back, but when it does you should see:

```
pod "hello-deployment-9794b4889-kms7p" deleted
```

If we then immediately list the pods, we see something interesting:

```
$ kubectl get pods
  NAME                                READY    STATUS    RESTARTS   AGE
  hello                               1/1      Running   0          33m
  hello-deployment-9794b4889-sx6jc    1/1      Running   0          9s
```

We see a new pod (in this case, "hello-deployment-9794b4889-sx6jc") was created and started by k8s for our hello deployment automatically! k8s did this because we instructed it that we wanted 1 replica pod to be running in the deployment's `spec` – this was the *desired* state – and when that didn't match the actual state (0 pods) k8s worked to change it. Remember, deployments are for programs that should *always be running*.

What do you expect to happen if we delete the original "hello" pod? Will k8s start a new one? Let's try it

```
$ kubectl delete pods hello
  pod "hello" deleted

$ kubectl get pods
  NAME                                READY    STATUS    RESTARTS   AGE
  hello-deployment-9794b4889-sx6jc    1/1      Running   0          4m
```

k8s did not start a new one. This "automatic self-healing" is one of the major difference between deployments and pods.

9.1.3 Scaling a Deployment

If we want to change the number of pods k8s runs for our deployment, we simply update the `replicas` attribute in our deployment file and apply the changes. Let's modify our "hello" deployment to run 4 pods. Modify `deployment-basic.yml` as follows:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 4
  selector:
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Apply the changes with:

```
$ kubectl apply -f deployment-basic.yml
  deployment.apps/hello-deployment configured
```

When we list pods, we see k8s has quickly implemented our requested change:

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-deployment-9794b4889-mk6qw    1/1     Running   0          11s
hello-deployment-9794b4889-sx6jc    1/1     Running   0          15m
hello-deployment-9794b4889-v2mb9    1/1     Running   0          11s
hello-deployment-9794b4889-vp6mp    1/1     Running   0          11s
```

9.1.4 EXERCISE

1) Delete several of the hello deployment pods and see what happens.

2) Scale the number of pods associated with the hello deployment back down to 1.

9.1.5 Updating Deployments with New Images

When we have made changes to the software or other aspects of a container image and we are ready to deploy the new version to k8s, we have to update the pods making up the corresponding deployment. We will use two different strategies, one for our "test" environment and one for "production".

Test Environments

A standard practice in software engineering is to maintain one or more "pre-production" environments, often times called "test" or "quality assurance" environments. These environments look similar to the "real" production environment where actual users will interact with the software, but few if any real users have access to them. The idea is that software developers can deploy new changes to a test environment and see if they work without the risk of potentially breaking the software for real users if they encounter unexpected issues.

Test environments are essential to maintaining quality software, and every major software project the Cloud and Interactive Computing group at TACC develops makes use of multiple test environments. We will have you create separate test and production environments as part of building the final project in this class.

It is also common practice to deploy changes to the test environment often, as soon as code is ready and tests are passing on a developer's laptop. We deploy changes to our test environments dozens of times a day while a large enterprise like Google may deploy millions of times a day. We will learn more about test environments and automated deployment strategies in the Continuous Integration section.

Image Management and Tagging

As you have seen, the `tag` associated with a Docker image is the string after the `:` in the name. For example, `` `ubuntu:18.04 `` has a tag of `18.04` representing the version of Ubuntu packaged in the image, while `jstubbs/hello-flask:dev` has a tag of `dev`, in this case indicating that the image was built from the `dev` branch of the corresponding git repository. Use of tags should be deliberate and is an important detail in a well designed software development release cycle.

Once you have created a deployment for a pod with a given image, there are two basic approaches to deploying an updated version of the container images to k8s:

1. Use a new image tag or
2. Use the same image tag and instruct k8s to download the image again.

Using new tags is useful and important whenever you may want to be able to recover or revert back to the previous image, but on the other hand, it can be tedious to update the tag every time there is a minor change to a software image.

Therefore, we suggest the following guidelines for image tagging:

1. During development when rapidly iterating and making frequent deployments, use a tag such as `dev` to indicate the image represents a development version of the software (and is not suitable for production) and simply overwrite the image tag with new changes. Instruct k8s to always try to download a new version of this tag whenever it creates a pod for the given deployment (see next section).
2. Once the primary development has completed and the code is ready for end-to-end testing and evaluation, begin to use new tags for each change. These are sometimes called "release candidates" and therefore, a tagging scheme such as `rc1`, `rc2`, `rc3`, etc., can be used for tagging each release candidate.

3. Once testing has completed and the software is ready to be deployed to production, tag the image with the version of the software. There are a number of different schemes for versioning software, such as Semantic Versioning (https://semver.org/), which will discuss later in the semester, time permitting.

ImagePullPolicy

When defining a deployment, we can specify an `ImagePullPolicy` which instructs k8s about when and how to download the image associated with the pod definition. For our test environments, we will instruct k8s to always try and download a new version of the image whenever it creates a new pod. We do this by specifying `imagePullPolicy: Always` in our deployment.

For example, we can add `imagePullPolicy: Always` to our hello-deployment as follows:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    app: hello-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-app
  template:
    metadata:
      labels:
        app: hello-app
    spec:
      containers:
        - name: hellos
          imagePullPolicy: Always
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

and now k8s will always try to download the latest version of `ubuntu:18.04` from Docker Hub every time it creates a new pod for this deployment. As discussed above, using `imagePullPolicy: Always` is nice during active development because you ensure k8s is always deploying the latest version of your code. Other possible values include `IfNotPresent` (the current default) which instructs k8s to only pull the image if it doesn't already exist on the worker node. This is the proper setting for a production deployment in most cases.

Deleting Pods to Update the Deployment

Note that if we have an update to our `:dev` image and we have set `imagePullPolicy: Always` on our deployment, all we have to do is delete the existing pods in the deployment to get the updated version deployed: as soon as we delete the pods, k8s will determine that an insufficient number of pods are running and try to start new ones. The `imagePullPolicy` instructs k8s to first try and download a newer version of the image.

9.1.6 Mounts, Volumes and Persistent Volume Claims

Some applications need access to storage where they can save data that will persist across container starts and stops. We saw how to solve this with Docker using a volume mount. In k8s, we use a combination of volume mounts, volumes and persistent volume claims.

Create a new file, `deployment-pvc.yml`, with the following contents, replacing "<username>" with your username:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-pvc-deployment
  labels:
    app: hello-pvc-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-pvc-app
  template:
    metadata:
      labels:
        app: hello-pvc-app
    spec:
      containers:
        - name: hellos
          image: ubuntu:18.04
          command: ['sh', '-c', 'echo "Hello, Kubernetes!" >> /data/out.txt && sleep␣
↪3600']
          volumeMounts:
          - name: hello-<username>-data
            mountPath: "/data"
      volumes:
      - name: hello-<username>-data
        persistentVolumeClaim:
          claimName: hello-<username>-data
```

We have added a `volumeMounts` stanza to `spec.containers` and we added a `volumes` stanza to the `spec`. These have the following effects:

- The `volumeMounts` include a `mountPath` attribute whose value should be the path in the container that is to be provided by a volume instead of what might possibly be contained in the image at that path. Whatever is provided by the volume will overwrite anything in the image at that location.
- The `volumes` stanza states that a volume with a given name should be fulfilled with a specific persistentVolumeClaim. Since the volume name (`hello-<username>-data`) matches the name in the `volumeMounts` stanza, this volume will be used for the volumeMount.
- In k8s, a persistent volume claim makes a request for some storage from a storage resource configured by the k8s administrator in advance. While complex, this system supports a variety of storage systems without requiring the application engineer to know details about the storage implementation.

Note also that we have changed the command to redirect the output of the `echo` command to the file `/data/out.txt`. This means that we should not expect to see the output in the logs for pod but instead in the file inside the container.

However, if we create this new deployment and then list pods we see something curious:

```
$ kubectl apply -f deployment-pvc.yml
$ kubectl get pods
  NAME                                    READY   STATUS    RESTARTS   AGE
  hello-deployment-9794b4889-mk6qw        1/1     Running   1          62m
  hello-deployment-9794b4889-sx6jc        1/1     Running   1          78m
  hello-deployment-9794b4889-v2mb9        1/1     Running   1          62m
  hello-deployment-9794b4889-vp6mp        1/1     Running   1          62m
  hello-pvc-deployment-74f985fffb-g9zd7   0/1     Pending   0          4m22s
```

Our "hello-deployment" pods are still running fine but our new "hello-pvc-deployment" pod is still in "Pending" status. It appears to be stuck. What could be wrong?

We can ask k8s to describe that pod to get more details:

```
$ kubectl describe pods hello-pvc-deployment-74f985fffb-g9zd7
  Name:           hello-pvc-deployment-74f985fffb-g9zd7
  Namespace:      designsafe-jupyter-stage
  Priority:       0
  Node:           <none>
  Labels:         app=hello-pvc-app
                  pod-template-hash=74f985fffb
  <... some output omitted ...>
  Tolerations:     node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                   node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
  Events:
    Type     Reason            Age    From               Message
    ----     ------            ----   ----               -------
    Warning  FailedScheduling  4m35s  default-scheduler  persistentvolumeclaim "hello-
↪jstubbs-data" not found
    Warning  FailedScheduling  4m35s  default-scheduler  persistentvolumeclaim "hello-
↪jstubbs-data" not found
```

At the bottom we see the "Events" section contains a clue: persistentvolumeclaim "hello-jstubbs-data" not found.

This is our problem. We told k8s to fill a volume with a persistent volume claim named "hello-jstubbs-data" but we never created that persistent volume claim. Let's do that now!

Open up a file called `hello-pvc.yml` and copy the following contents, being sure to replace `<username>` with your TACC username:

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: hello-<username>-data
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: rbd
  resources:
    requests:
      storage: 1Gi
```

We will use this file to create a persistent volume claim against the storage that has been set up in the TACC k8s cluster. In order to use this storage, you do need to know the storage class (in this case, "rbd", which is the storage class for utilizing the Ceph storage system), and how much you want to request (in this case, just 1 Gig), but you don't need to know how the storage was implemented.

We create this pvc object with the usual `kubectl apply` command:

```
$ kubectl apply -f hello-pvc.yml
  persistentvolumeclaim/hello-jstubbs-data created
```

Great, with the pvc created, let's check back on our pods:

```
$ kubectl get pods
  NAME                                    READY   STATUS        RESTARTS   AGE
  hello-deployment-9794b4889-mk6qw        1/1     Running       46         46h
  hello-deployment-9794b4889-sx6jc        1/1     Running       46         46h
  hello-deployment-9794b4889-v2mb9        1/1     Running       46         46h
  hello-deployment-9794b4889-vp6mp        1/1     Running       46         46h
  hello-pvc-deployment-ff5759b64-sc7dk    1/1     Running       0          45s
```

Like magic, our "hello-pvc-deployment" now has a running pod without us making any additional API calls to k8s! This is the power of the declarative aspect of k8s. When we created the hello-pvc-deployment, we told k8s to always keep one pod with the properties specified running at all times, if possible, and k8s continues to try and implement our wishes until we instruct it to do otherwise.

Note: You cannot scale a pod with a per

9.1.7 Exec Commands in a Running Pod

Because the command running within the "hello-pvc-deployment" pod redirected the echo statement to a file, the hello-pvc-deployment-ff5759b64-sc7dk will have no logs. (You can confirm this is the case for yourself using the `logs` command as an exercise).

In cases like these, it can be helpful to run additional commands in a running pod to explore what is going on. In particular, it is often useful to run shell in the pod container.

In general, one can run a command in a pod using the following:

```
$ kubectl exec <options> <pod_name> -- <command>
```

To run a shell, we will use:

```
$ kubectl exec -it <pod_name> -- /bin/bash
```

The `-it` flags might look familiar from Docker – they allow us to "attach" our standard input and output to the command we run in the container. The command we want to run is `/bin/bash` for a shell.

Let's exec a shell in our "hello-pvc-deployment-ff5759b64-sc7dk" pod and look around:

```
$ k exec -it  hello-pvc-deployment-5b7d9775cb-xspn7 -- /bin/bash
  root@hello-pvc-deployment-5b7d9775cb-xspn7:/#
```

Notice how the shell prompt changes after we issue the `exec` command – we are now "inside" the container, and our prompt has changed to "root@hello-pvc-deployment-5b7d9775cb-xspn" to indicate we are the root user within the container.

Let's issue some commands to look around:

```
$ pwd
  /
  # cool, exec put us at the root of the container's file system
```

(continues on next page)

(continued from previous page)

```
$ ls -l
  total 8
  drwxr-xr-x   2 root root 4096 Jan 18 21:03 bin
  drwxr-xr-x   2 root root    6 Apr 24  2018 boot
  drwxr-xr-x   3 root root 4096 Mar  4 01:06 data
  drwxr-xr-x   5 root root  360 Mar  4 01:12 dev
  drwxr-xr-x   1 root root   66 Mar  4 01:12 etc
  drwxr-xr-x   2 root root    6 Apr 24  2018 home
  drwxr-xr-x   8 root root   96 May 23  2017 lib
  drwxr-xr-x   2 root root   34 Jan 18 21:03 lib64
  drwxr-xr-x   2 root root    6 Jan 18 21:02 media
  drwxr-xr-x   2 root root    6 Jan 18 21:02 mnt
  drwxr-xr-x   2 root root    6 Jan 18 21:02 opt
  dr-xr-xr-x 887 root root    0 Mar  4 01:12 proc
  drwx------   2 root root   37 Jan 18 21:03 root
  drwxr-xr-x   1 root root   21 Mar  4 01:12 run
  drwxr-xr-x   1 root root   21 Jan 21 03:38 sbin
  drwxr-xr-x   2 root root    6 Jan 18 21:02 srv
  dr-xr-xr-x  13 root root    0 May  5  2020 sys
  drwxrwxrwt   2 root root    6 Jan 18 21:03 tmp
  drwxr-xr-x   1 root root   18 Jan 18 21:02 usr
  drwxr-xr-x   1 root root   17 Jan 18 21:03 var
  # as expected, a vanilla linux file system.
  # we see the /data directory we mounted from the volume...

$ ls -l data/out.txt
  -rw-r--r-- 1 root root 19 Mar  4 01:12 data/out.txt
  # and there is out.txt, as expected

$ cat data/out.txt
  Hello, Kubernetes!
  # and our hello message!

$ exit
  # we're ready to leave the pod container
```

Note: To exit a pod from within a shell (i.e., `/bin/bash`) type "exit" at the command prompt.

Note: The `exec` command can only be used to execute commands in *running* pods.

9.1.8 Persistent Volumes Are... Persistent

The point of persistent volumes is that they live beyond the length of one pod. Let's see this in action. Do the following:

1. Delete the "hello-pvc" pod. What command do you use?
2. After the pod is deleted, list the pods again. What do you notice?
3. What contents do you expect to find in the `/data/out.txt` file? Confirm your suspicions.

Solution.

```
$ kubectl delete pods hello-pvc-deployment-5b7d9775cb-xspn7
  pod "hello-pvc-deployment-5b7d9775cb-xspn7" deleted

$ kubectl get pods
  NAME                                    READY   STATUS              RESTARTS   AGE
  hello-deployment-9794b4889-mk6qw        1/1     Running             47         47h
  hello-deployment-9794b4889-sx6jc        1/1     Running             47         47h
  hello-deployment-9794b4889-v2mb9        1/1     Running             47         47h
  hello-deployment-9794b4889-vp6mp        1/1     Running             47         47h
  hello-pvc-deployment-5b7d9775cb-7nfhv   0/1     ContainerCreating   0          46s
  # wild -- a new hello-pvc-deployment pod is getting created automatically!

# let's exec into the new pod and check it out!
$ k exec -it hello-pvc-deployment-5b7d9775cb-7nfhv -- /bin/bash

$ cat /data/out.txt
  Hello, Kubernetes!
  Hello, Kubernetes!
```

Warning: Deleting a persistent volume claim deletes all data contained in all volumes filled by the PVC permanently! This cannot be undone and the data cannot be recovered!

9.1.9 Additional Resources

- Kubernetes Deployments Documentation
- Persistent Volumes
- Ceph RBD Storage class in k8s

9.2 Services

Services are the k8s resource one uses to expose HTTP APIs, databases and other components that communicate on a network to other k8s pods and, ultimately, to the outside world. To understand services we need to first discuss how k8s networking works.

9.2.1 k8s Networking Overview

Note: We will be covering just the basics of k8s networking, enough for you to become proficient with the main concepts involved in deploying your application. Many details and advanced concepts will be omitted.

k8s creates internal networks and attaches pod containers to them to facilitate communication between pods. For a number of reasons, including security, these networks are not reachable from outside k8s.

Recall that we can learn the private network IP address for a specific pod with the following command:

```
$ kubectl get pods <pod_name> -o wide
```

For example:

```
$ kubectl get pods hello-deployment-9794b4889-mk6qw -o wide
  NAME                                READY   STATUS    RESTARTS   AGE   IP
↪ NODE   NOMINATED NODE   READINESS GATES
  hello-deployment-9794b4889-mk6qw   1/1     Running   277        11d   10.244.3.176
↪ c01    <none>           <none>
```

This tells us k8s assigned an IP address of `10.244.3.176` to our hello-deployment pod.

k8s assigns every pod an IP address on this private network. Pods that are on the same network can communicate with other pods using their IP address.

9.2.2 Ports

To communicate with a program running on a network, we use ports. We saw how our flask program used port 5000 to communicate HTTP requests from clients. We can expose ports in our k8s deployments by defining a `ports` stanza in our `template.spec.containers` object. Let's try that now.

Create a file called `hello-flask-deployment.yml` and copy the following contents

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloflask
  labels:
    app: helloflask
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helloflask
  template:
    metadata:
      labels:
        app: helloflask
    spec:
      containers:
        - name: helloflask
          imagePullPolicy: Always
          image: jstubbs/hello-flask
          env:
          - name: FLASK_APP
            value: "app.py"
          ports:
          - name: http
            containerPort: 5000
```

Much of this will look familiar. We are creating a deployment that matches the pod description given in the `template.spec` stanza. The pod description uses an image, `jstubbs/hello-flask`. This image runs a very simple flask server that responds with simple text messages to a few endpoints.

The `ports` attribute is a list of k8s port descriptions. Each port in the list includes:

- `name` – the name of the port, in this case, `http`. This could be anything we want really.
- `containerPort` – the port inside the container to expose, in this case `5000`. This needs to match the port that the containerized program (in this case, flask server) is binding to.

Let's create the helloflask deployment using `kubectl apply`

```
$ kubectl apply -f hello-flask-deployment.yml
  deployment.apps/hello-flask-deployment configured
```

With our deployment created, we should see a new pod.

Exercise. Determine the IP address of the new pod for the hello-flask-deployment.

Solution.

```
$ kubectl get pods
  NAME                                     READY   STATUS    RESTARTS   AGE
  hello-deployment-9794b4889-w4jlq         1/1     Running   0          56m
  hello-pvc-deployment-6dbbfdc4b4-sxk78    1/1     Running   231        9d
  helloflask-86d4c7d8db-2rkg5              1/1     Running   0          5m10s


$ kubectl get pods helloflask-86d4c7d8db-2rkg5 -o wide
  NAME                           READY   STATUS    RESTARTS   AGE     IP           ␣
↪NODE   NOMINATED NODE   READINESS GATES
  helloflask-86d4c7d8db-2rkg5   1/1     Running   0          6m27s   10.244.7.95  ␣
↪c05    <none>           <none>

# Therefore, the IP address is 10.244.7.95
```

We found the IP address for our flask container, but if we try to communicate with it from the k8s API node, we will get an error:

```
$ curl 10.244.7.95:5000
  curl: (7) Failed connect to 10.244.7.95:5000; Network is unreachable
```

This is because the 10.244.*.* private k8s network is not available from the outside, not even from the API node. However, it *is* available from other pods in the namespace.

9.2.3 A Debug Deployment

For exploring and debugging k8s deployments, it can be helpful to have a basic container on the network. We can create a deployment for this purpose.

For example, let's create a deployment using the official python 3.9 image. We can run a sleep command inside the container as the primary command, and then, once the container pod is running, we can use `exec` to launch a shell inside the container.

EXERCISE

1. Create a new "debug" deployment using the following definition:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: py-debug-deployment
  labels:
    app: py-app
spec:
```

(continues on next page)

(continued from previous page)

```
  replicas: 1
  selector:
    matchLabels:
      app: py-app
  template:
    metadata:
      labels:
        app: py-app
    spec:
      containers:
        - name: py39
          image: python:3.9
          command: ['sleep', '999999999']
```

(Hint: paste the content into a new file called `deployment-python-debug.yml` and then use the `kubectl apply` command).

2. Exec into the running pod for this deployment. (Hint: find the pod name and then use the `kubectl exec` command, running the shell (`/bin/bash`) command in it).

Once we have a shell running inside our debug deployment pod, we can try to access our flask server. Recall that the IP and port for the flask server were determined above to be 10.244.7.95:5000 (yours will be different).

If we try to access it using curl from within the debug container, we get:

```
root@py-debug-deployment-5cc8cdd65f-xzhzq: $ curl 10.244.7.95:5000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually␣
↪please check your spelling and try again.</p>
```

That's a different error from before, and that's good! This time, the error is from flask, and it indicates that flask doesn't have a route for the root path (`/`).

The `jstubbs/hello-flask` image does not define a route for the root path (`/`) but it does define a route for the path `/hello-service`. If we try that path, we should get a response:

```
root@py-debug-deployment-5cc8cdd65f-xzhzq: $ curl 10.244.7.95:5000/hello-service
Hello world
```

Great! k8s networking from within the private network is working as expected!

9.2.4 Services

We saw above how pods can use the IP address of other pods to communicate. However, that is not a great solution because we know the pods making up a deployment come and go. Each time a pod is destroyed and a new one created it gets a new IP address. Moreover, we can scale the number of replica pods for a deployment up and down to handle more or less load.

How would an application that needs to communicate with a pod know which IP address to use? If there are 3 pods comprising a deployment, which one should it use? This problem is referred to as the *service discovery problem* in distributed systems, and k8s has a solution for it.. the `Service` abstraction.

A k8s service provides an abstract way of exposing an application running as a collection of pods on a single IP address and port. Let's define a service for our hello-flask deployment.

Copy and paste the following code into a file called `hello-flask-service.yml`:

```
---
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  type: ClusterIP
  selector:
    app: helloflask
  ports:
  - name: helloflask
    port: 5000
    targetPort: 5000
```

Let's look at the `spec` description for this service.

- `type` – There are different types of k8s services. Here we are creating a `ClusterIP` service. This creates an IP address on the private k8s network for the service. We may see other types of k8s services later.
- `selector` – This tells k8s what pod containers to match for the service. Here we are using a label, `app: helloflask`, which means k8s will link all pods with this label to our service. Note that it is important that this label match the label applied to our pods in the deployment, so that k8s links the service up to the correct pods.
- `ports` - This is a list of ports to expose in the service.
- `ports.port` – This is the port to expose on the service's IP. This is the port clients will use when communicating via the service's IP address.
- `ports.targetPort` – This is the port on the pods to target. This needs to match the port specified in the pod description (and the port the containerized program is binding to).

We create this service using the `kubectl apply` command, as usual:

```
$ kubectl apply -f hello-flask-service.yml
  service/hello-service configured
```

We can list the services:

```
$ kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
↪     AGE
hello-service   ClusterIP   10.108.58.137    <none>        5000/TCP
↪     4s
```

We see k8s created a new service with IP `10.108.58.137`. We should be able to use this IP address (and port 5000) to communicate with our flask server. Let's try it. Remember that we have to be on the k8s private network, so we need to exec into our debug deployment pod first.

```
$ kubectl exec -it py-debug-deployment-5cc8cdd65f-xzhzq -- /bin/bash

# from inside the container ---
root@py-debug-deployment-5cc8cdd65f-xzhzq:/ $ curl 10.108.58.137:5000/hello-service
Hello world
```

It worked! Now, if we remove our hello-flask pod, k8s will start a new one with a new IP address, but our service will automatically route requests to the new pod. Let's try it.

```
# remove the pod ---
$ kubectl delete pods helloflask-86d4c7d8db-2rkg5
  pod "helloflask-86d4c7d8db-2rkg5" deleted

# see that a new one was created ---
$ kubectl get pods
  NAME                                     READY   STATUS    RESTARTS   AGE
  hello-deployment-9794b4889-w4jlq         1/1     Running   2          175m
  hello-pvc-deployment-6dbbfdc4b4-sxk78    1/1     Running   233        9d
  helloflask-86d4c7d8db-vbn4g              1/1     Running   0          62s

# it has a new IP ---
$ kubectl get pods helloflask-86d4c7d8db-vbn4g -o wide
  NAME                             READY   STATUS    RESTARTS   AGE    IP
↪NODE   NOMINATED NODE   READINESS GATES
  helloflask-86d4c7d8db-vbn4g   1/1     Running   0          112s   10.244.7.96   c05
↪    <none>           <none>
# Yep, 10.244.7.96 -- that's different; the first pod had IP 10.244.7.95

# but back in the debug deployment pod, check that we can still use the service IP --
root@py-debug-deployment-5cc8cdd65f-xzhzq:/ $ curl 10.108.58.137:5000/hello-service
Hello world
```

Note that k8s is doing something non-trivial here. Each pod could be running on one of any number of worker computers in the TACC k8s cluster. When the first pod was deleted and k8s created the second one, it is quite possible it started it on a different machine. So k8s had to take care of rerouting requests from the service to the new machine.

k8s can be configured to do this "networking magic" in different ways. While the details are beyond the scope of this course, keep in mind that the virtual networking that k8s uses does come at a small cost. For most applications, including long-running web APIs and databases, this cost is negligible and isn't a concern. But for high-performance applications, and in particular, applications whose performance is bounded by the performance of the underlying network, the overhead can be significant.

9.2.5 HomeWork 6 – Deploying Our Flask API to k8s

In this section we will use class time to deploy our Flask API to k8s. This will be a guided, hands-on lab, and it will also be submitted for a grade as HomeWork 5. Feel free to ask questions as you work through the lab. Any thing left

Our goal today is to create a "test" environment for our Flask API application. We will be using names and labels accordingly. Later in the semester, you will create a "production environment for your Flask API application as well. You can use this guide to do that.

In each step you will create a k8s object described in a separate yml file. Name the files `<username>-<env>-<app>-<kind>.yml`, Use "test" for `<env>` since we are creating the test environment. For example, my Redis deployment would `jstubbs-test-redis-deployment.yml` while my redis service would be called `jstubbs-test-redis-service.yml`.

Step 1. We will start by focusing on our Redis container. Our Flask API depends on Redis so it makes sense to start there. Since Redis writes our application data to disk, we will need a way to save the data independent of the Redis pods. Create a persistent volume claim for your Redis data. Use the following information when creating your PVC:

- The name of your PVC should include your TACC username and the word "test", to indicate it is in the test environment.
- We'll make use of `labels` to add additional metadata to our k8s objects that will help us search and filter them. Let's add a `username` label and an `env` label. The value for `username` should be your tacc username and the value for `env` should be `test`, to indicate that this is the test environment.

- The `accessModes` should include a single entry, `readWriteOnce`.
- The `storageClassName` should be `rbd`.
- Be sure to request 1 GB (`1Gi`) of storage.

Step 2. Create a deployment for the Redis database. Be sure to include the following:

- The name of your redis deployment should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- Be sure to set `replicas: 1` as Redis is a stateful application.
- For the image, use `redis:5.0.0`; you do not need to set a command.
- Add the `username` and `env` lables to the pod as well. Also add an `app` label with value `<username>-test-redis`. This will be important in the next step.
- Be sure to create a `volumeMount` and associate it with a `volume` that is filled by the PVC you created in Step 1. For the mount path, use `/data`, as this is where Redis writes its data.

Step 3. Create a service for your Redis database. This will give you a persistent IP address to use to talk to Redis, regardless of the IPs that may be assigned to individual Redis pods. Be sure to include the following:

- The name of your redis service should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- The `type` of service should be `ClusterIP`.
- Define a `selector` that will select your Redis pods and only your redis pods. What label should you use? Hint: the `env` and `username` labels won't be unique enough.
- Make sure `port` and `targetPort` match the Redis port.

Once you are done with Steps 1 though 3, check your work:

- Look up the service IP address for your test redis service.
- Exec into a Python debug container.
- Install the redis python library.
- Launch the python shell and import redis
- Create a Python redis client object using the IP and port of the service, something like: `rd = redis.StrictRedis(host='10.101.101.139', port=6379, db=0)`
- Create a key and make sure you can get the key.
- In another shell on isp02, delete the redis pod. Check that k8s creates a new redis pod.
- Back in your python shell, check that you can still get the key using the same IP. This will show that your service is working and that your Redis database is persisting data to the PVC (i.e., the data are surviving pod restarts).

Step 4. Create a deployment for your flask API. If it helps, you can use your Redis deployment as a starting point. Be sure to:

- The name of your flask service should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- start 2 replicas of your flask API pod.

- Be sure to expose port 5000.

Step 5. Create a service for your flask API. This will give you a persistent IP address to use to talk to your flask API, regardless of the IPs that may be assigned to individual flask API pods. Be sure to include the following:

- The name of your redis service should include your TACC username and the word "test", to indicate it is in the test environment.
- Use the same `username` and `env` labels for both the deployment and the pod template.
- The `type` of service should be `ClusterIP`.
- Define a `selector` that will select your flask API pods and only your flask API pods.
- Make sure `port` and `targetPort` match the flask port.

9.3 k8s Cheat Sheet

This all-in-one k8s cheat sheet can be used for quick reference.

9.3.1 k8s Resource Types

Here are the primary k8s resource types we have covered in this class:

- `Pods` – Pods are the simplest unit of compute in k8s and represent a generalization of the Docker container. Pods can contain more than one container, and every container within a pod is scheduled together, on the same machine, with a single IP address and shared file system. Pod definitions include a `name`, a (Docker) `image`, a `command` to run, as well as an `ImagePullPolicy`, `volumeMounts` and a set of `ports` to expose. Pods can be thought of as disposable and temporary.
- `Deployments` – Deployments are the k8s resource type to use for deploying *long-running* application components, such as APIs, databases, and long-running worker programs. Deployments are made up of one or more matching pods, and the matching is done using `labels` and `labelSelectors`.
- `PersistentVolumeClaims` (PVCs) – PVCs create a named storage request against a storage class available on the k8s cluster. PVCs are used to fill volumes with permenant storage so that data can be saved across different pod executions for the same stateful application (e.g., a database).
- `Services` – Services are used to expose an entire application component to other pods. Services get their own IP address which persists beyond the life of the individual pods making up the application.

9.3.2 kubectl Commands

Here we collect some of the most commonly used `kubectl` commands for quick reference.

Command	Description	Example
kubectl get <resource_type>	List all objects of a given resource type.	kubectl get pods
kubectl get <type> <name>	Get one object of a given type by name.	kubectl get pods hello-pod
kubectl get <type> <name> -o wide	Show additional details of an object	kubectl get pods hello-pod -o wide
kubectl describe <type> <name>	Get full details of an object by name.	kubectl describe pods hello-pod
kubectl logs <name>	Get the logs of a running pod by name.	kubectl logs hello-pod
kubectl logs -f <name>	Tail the logs of a running pod by name.	kubectl logs -f hello-pod
kubectl logs –since <time> <name>	Get the logs of a running pod since a given time.	kubectl logs –since 1m -f hello-pod
kubectl exec -it <name> – <cmd>	Run a command, <cmd>, in a running pod.	kubectl exec -it hello-pod – /bin/bash
kubectl apply -f <file>	Create or update an object description using a file.	kubectl apply -f hello-pod.yml

CHAPTER

TEN

WEEK 11: ASYNCHRONOUS PROGRAMMING

In this week, we will begin our discussion of concurrency and asynchronous programming with the goal of covering the material you you will need to add a "long running" jobs functionality to your Flask API project.

10.1 Concurrency and Queues

10.1.1 Motivation

Our Flask API is useful because it can return information about objects in our database, and in general, looking up or storing objects in the database is a very "fast" operation, on the order of a few 10s of milliseconds. However, many interesting and useful operations are not nearly as quick to perform. They are many examples from both research computing and industrial computing where the computations take much longer; for example, on the order of minutes, hours, days or even longer.

Examples include:

- Aligning a set of genomic sequence fragments to a reference genome
- Executing a large mathematical model simulating galaxy formation
- Running the payroll program at the end of the month to send checks to all employees of a large enterprise.
- Sending a "welcome back" email to every student enrolled at the university at the start of the semester.

We want to be able to add functionality like this to our API system. We'd like to provide a new API endpoint where a user could describe some kind of long-running computation to be performed and have our system perform it for them. But there are a few issues:

- The HTTP protocol was not built for long-running tasks, and most programs utilizing HTTP expect responses "soon", on the order of a few seconds. Many programs have hard timeouts around 30 or 60 seconds.
- The networks on which HTTP connections are built can be interrupted (even just briefly) over long periods of time. If a connection is severed, even for a few milliseconds, what happens to the long-running computation?
- Long-running tasks like the ones above can be computationally intensive and require a lot of computing resources. If our system becomes popular (even with a single, enthusiastic user), we may not be able to keep up with demand. We need to be able to throttle the number of computations we do.

To address these challenges, we will implement a "Jobs API" as an *asynchronous* endpoint. Over the next few lectures, we will spell out precisely what this means, but for now, we'll give a quick high-level overview as motivation. Don't worry about understanding all the details here.

10.1.2 Jobs API – An Introduction

The basic idea is that we will have a new endpoint in our API at a path `/jobs` (or something similar). A user wanting to have our system perform a long-running task will create a new job. We will use RESTful semantics, so the user will create a new job by making an HTTP POST request to `/jobs`, describing the job in the POST message body (in JSON).

However, instead of performing the actual computation, the Jobs API will simply record that the user has requested such a computation. It will store that in Redis and immediately respond to the user. So, the response will not include the result of the job itself but instead it will indicate that the request has been received and it will be worked on in due time. Also, and critically, it will provide an `id` for the job that the user can use to check the status later and, eventually, get the actual result.

So, in summary:

1. User makes an HTTP POST to `/jobs` to create a job.
2. Jobs API validates that the job is a valid job, creates an `id` for it, and stores the job description with the `id` in Redis.
3. Jobs API responds to the user immediately with the `id` of the job it generated.
4. In the background, *some other python program* we write (referred to as a "worker") will, at some point in the future, actually start the job and monitor it to completion.

This illustrates the *asynchronous* and *concurrent* nature of our Jobs API, terms we will define precisely in the sequel. Intuitively, you can probably already imagine what we mean here – multiple jobs can be worked on at the same time by different instances of our program (i.e., different workers), and the computation happens asynchronously from the original user's request.

10.1.3 Concurrency and Queues

A computer system is said to be *concurrent* if multiple agents or components of the system can be in progress at the same time without impacting the correctness of the system.

While components of the system are in progress at the same time, the individual operations themselves may happen sequentially. In general, a system being concurrent means that the different components can be executed at the same time or in different orders without impacting the overall correctness of the system.

There are many techniques for making programs concurrent; we will primarily focus on a technique that leverages the *queue* data structure. But first, an example.

A First Example

Suppose we want to build a system for maintaining the balance of a bank account where multiple agents are acting on the account (withdrawing and/or depositing funds) at the same time. We will consider two different approaches.

Approach 1. Whenever an agent receives an order to make a deposit or withdraw, the agent does the following steps:

1. Makes a query to determine the current balance.
2. Computes the new balance based on the deposit or withdraw amount.
3. Makes a query to update the balance to the computed amount.

This approach is not concurrent because the individual operations of different agents cannot be reordered.

For example, suppose we have:

- Two agents, agent A and agent B, and a starting balance of $50.

- Agent A gets an order to deposit $25 at the same time that agent B gets an order to withdraw $10.

In this case, the final balance should be $65 (=$50 + $25 - $10).

The system will arrive at this answer as long as steps 1, 2 and 3 for one agent are done before any steps for the other agent are started; for ex, A1, A2, A3, B1, B2, B3.

However, if the steps of the two agents are mixed then the system will not arrive at the correct answer.

For example, suppose the steps of the two agents were performed in this order: A1, A2, B1, B2, A3, B3. What would the final result be? The listing below shows what each agents sees at each step.

- A1. Agent A determines the current balance to be $50.
- A2. Agent A computes a new balance of $50 + $25 = $75.
- B1. Agent B determines the current balance to be $50.
- B2. Agent B computes a new balance of $50 - $10 = $40.
- A3. Agent A updates the balance to be $75.
- B3. Agent B updates the balance to be $40.

In this case, the system will compute the final balance to be $40! Hopefully this is not your account! :)

We will explore an alternative approach that is concurrent, but to do that we first need to introduce the concept of a queue.

Queues

A queue is data structure that maintains an ordered collection of items. The queue typically supports just two operations:

- Enqueue (aka "put") - add a new item to the queue.
- Dequeue (aka "get") - remove an item from the queue.

Items are removed from a queue in First-In-First-Out (FIFO) fashion: that is, the item removed from the first dequeue operation will be the first item added to the queue, the item removed from the second dequeue operation will be the second item added to the queue, and so on.

Sometimes queues are referred to as "FIFO Queues" for emphasis.

Basic Queue Example

Consider the set of (abstract) operations on a Queue object.

```
1. Enqueue 5
2. Enqueue 7
3. Enqueue A
4. Dequeue
5. Enqueue 1
6. Enqueue 4
7. Dequeue
8. Dequeue
```

The order of items returned is:

```
5, 7, A
```

And the contents of the Queue after Step 8 is

```
1, 4
```

In-memory Python Queues

The Python standard library provides an in-memory Queue data structure via its `queue` module. To get started, import the `queue` module and instantiate a `queue.Queue` object:

```
>>> import queue
>>> q = queue.Queue()
```

The Python Queue object has the following features:

- The `q` object supports `.put()` and `.get()` to put a new item on the queue, and get an item off the queue, respectively
- `q.put()` can take an arbitrary Python object and `q.get()` returns a Python object from the queue.

Let's perform the operations above using the `q` object.

Exercise. Use a series of `q.put()` and `q.get()` calls to perform Steps 1-8 above. Verify the the order of items returned.

Exercise. Verify that arbitrary Python objects can by put onto and retrieved from the queue by inserting a list and a dictionary.

Queues are a fundamental ingredient in concurrent programming, a topic we will turn to next.

A Concurrent Approach to Our Example

Approach 2. Whenever an agent receives an order to make a withdraw or deposit, the agent simply writes the order to a queue; a positive number indicates a deposit while a negative number indicates a withdraw. The account system keeps a running "balancer" agent whose only job is to read items off the queue and update the balance.

This approach is concurrent because the order of the agents' steps can be mixed without impacting the overall result. This fact essentially comes down to the commutativity of addition and subtraction operations: i.e., `50 + 25 - 10 = 50 - 10 + 25`.

Note that the queue of orders could be generalized to a "queue of tasks" (transfer some amount from account A to account B, close account C, etc.).

Queues in Redis

The Python in-memory queues are very useful for a single Python program, but we ultimately want to share queues across multiple Python programs/containers.

The Redis DB we have been using can also be used to provide a queue data structure for clients running in different containers. The basic idea is:

- Use a Redis list data structure to hold the items in the queue.
- Use the Redis list operations `rpush`, `lpop`, `llen`, etc. to create a queue data structure.

For example:

- `rpush` will add an element to the end of the list.
- `lpop` will return an element from the front of the list, and return nothing if the list is empty.

- `llen` will return the number of elements in the list.

Fortunately, we don't have to implement the queue ourselves, but know that if we needed to we could without too much effort.

Using the hotqueue library

We will leverage a small, open source Python library called `hotqueue` which has already implemented the a Queue data structure in Redis using the approach outlined above. Besides not having to write it ourselves, the use of `hotqueue` will afford us a few additional features which we will look at later.

Here are the basics of the `hotqueue` library:

- Hotqueue is not part of the Python standard library; you can install it with `pip install hotqueue`
- Creating a new queue data structure or connecting to an existing queue data structure is accomplished by creating a `HotQueue` object.
- Constructing a `HotQueue` object takes very similar parameters to that of the `StrictRedis` but also takes a `name` attribute. The `HotQueue` object ultimately provides a connection to the Redis server.
- Once constructed, a `HotQueue` object has `.put()` and `.get()` methods that act just like the corresponding methods of an in-memory Python queue.

A Hotqueue Example

We will work this example out on the k8s cluster. You will need a Redis pod running on the cluster and you will also need the python debug pod you created last lecture.

If you prefer, you can create a new deployment that uses the `jstubbs/redis-client` image with the required libraries already installed installed using the following code –

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-client-debug-deployment
  labels:
    app: redis-client-debug
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis-client-debug
  template:
    metadata:
      labels:
        app: redis-client-debug
    spec:
      containers:
        - name: py39
          image: jstubbs/redis-client
          command: ['sleep', '999999999']
```

With your debug pod running, first, exec into it and install `redis` and `hotqueue`. You can optionally also install `ipython` which is a nicer Python REPL (Read, Evaluate, Print Loop).

Note: The `jstubbs/redis-client` image has these libraries already installed.

```
$ kubectl get pods -o wide
  NAME                                      READY   STATUS    RESTARTS   AGE    IP
↪        NODE   NOMINATED NODE   READINESS GATES
  hello                                     1/1     Running   199        8d     10.244.
↪5.214   c04    <none>           <none>
  hello-deployment-55f4459bf-npdrm          1/1     Running   79         3d7h   10.244.
↪5.5     c04    <none>           <none>
  hello-pvc-deployment-6dbbfdc4b4-whjwb     1/1     Running   31         31h    10.244.
↪3.143   c01    <none>           <none>
  helloflask-848c4fb54f-9j4fd               1/1     Running   0          30h    10.244.
↪3.188   c01    <none>           <none>
  helloflask-848c4fb54f-gpqhb               1/1     Running   0          30h    10.244.
↪5.75    c04    <none>           <none>
  jstubbs-test-redis-64cbc6b8cf-f6qrl       1/1     Running   0          3m5s   10.244.
↪3.237   c01    <none>           <none>
  py-debug-deployment-5cc8cdd65f-tr9gq      1/1     Running   0          31h    10.244.
↪3.177   c01    <none>           <none>

$ kubectl exec -it py-debug-deployment-5cc8cdd65f-tr9gq -- /bin/bash

$ pip install redis hotqueue ipython
```

Start the python (or ipython) shell and create the `hotQueue.Queue` object. You can use the Redis IP directly, or use the Redis service IP if you creates one.

```
>>> from hotqueue import HotQueue
>>> q = HotQueue("queue", host="<Redis_IP>", port=6379, db=1)
```

Note how similar the `HotQueue()` instantiation is to the `StrictRedis` instantiation. In the example above we named the queue `queue` (not too creative), but it could have been anything.

Note: In the definition above, we have set `db=1` to ensure we don't interfering with the main data of your Flask app.

Now we can add elements to the queue using the *.put()*; just like with in-memory Python queues, we can put any Python object into the queue:

```
>>> q.put(1)
>>> q.put('abc')
>>> q.put(['1', 2, {'key': 'value'}, '4'])
```

We can check the number of items in queue at any time using the *len* built in:

```
>>> len(q)
3
```

And we can remove an item with the *.get()* method; remember - the queue follows a FIFO principle:

```
>>> q.get()
1
>>> len(q)
2
>>> q.get()
```

(continues on next page)

(continued from previous page)

```
'abc'
>>> len(q)
1
```

Under the hood, the `hotqueue.Queue` is just a Redis object, which we can verify using a redis client:

```
>>> import redis
>>> rd = redis.StrictRedis(host="<Redis IP>", port=6379, db=1)
>>> rd.keys()
[b'hotqueue:queue']
```

Note that the queue is just a single key in the Redis server `(db=1)`.

And just like with other Redis data structures, we can connect to our queue from additional Python clients and see the same data.

Exercise. In a second SSH shell, scale your Python debug deployment to 2 replicas, install redis, hotqueue, and ipython in the new replica, start iPython and connect to the same queue. Prove that you can use get and put to "communicate" between your two Python programs.

10.2 Messaging Systems

The Queue is a powerful data structure which forms the foundation of many concurrent design patterns. Often, these design patterns center around passing messages between agents within the concurrent system. We will explore one of the simplest and most useful of these message-based patterns - the so-called "Task Queue". Later, we may also look at the somewhat related "Publish-Subscribe" pattern (also sometimes referred to as "PubSub").

10.2.1 Task Queue (or Work Queue)

In a task queue system,

- Agents called "producers" write messages to a queue that describe work to be done.
- A separate set of agents called "consumers" receive the messages and do the work. While work is being done, no new messages are received by the consumer.
- Each message is delivered exactly once to a single consumer to ensure no work is "duplicated".
- Multiple consumers can be processing "work" messages at once, and similarly, 0 consumers can be processing messages at a given time (in which case, messages will simply queue up).

The Task Queue pattern is a good fit for our jobs service.

- Our Flask API will play the role of producer.
- One or more "worker" programs will play the role of consumer.
- Workers will receive messages about new jobs to execute and performing the analysis steps.

10.2.2 Task Queues in Redis

The `HotQueue` class provides two methods for creating a task queue consumer; the first is the `.consume()` method and the second is the `q.worker` decorator.

The Consume Method

With a `q` object defined like `q = HotQueue("some_queue", host="<Redis_IP>", port=6379, db=1)`, the consume method works as follows:

- The `q.consume()` method returns an iterator which can be looped over using a `for` loop (much like a list).
- The `q.consume()` method blocks (i.e., waits indefinitely) when there are no additional messages in the queue named `some_queue`.

The basic syntax of the consume method is this:

```
for item in q.consume():
    # do something with item
```

Exercises. Complete the following:

1. Start/scale two python debug containers with redis and hotqueue installed (you can use the `jstubbs/redis-client` image if you prefer). In two separate shells, exec into each debug container and start ipython.
2. In each terminal, create a `HotQueue` object pointing to the same Redis queue.
3. In the first terminal, add three or four Python strings to the queue; check the length of the queue.
4. In the second terminal, use a `for` loop and the `.consume()` method to print objects in the queue to the screen.
5. Observe that the strings are printed out in the second terminal.
6. Back in the first terminal, check the length of the queue; add some more objects to the queue.
7. Confirm the newly added objects are "instantaneously" printed to the screen back in the second terminal.

The q.worker Decorator

Given a Hotqueue queue object, `q`, the `q.worker` decorator is a convenience utility to turn a function into a consumer without having to write the for loop. The basic syntax is:

```
@q.worker
def do_work(item):
    # do something with item
```

In the example above, `item` will be populated with the item dequeued.

Then, to start consuming messages, simply call the function:

```
>>> do_work()
# ... blocks until new messages arrive
```

Note: The `@q.worker` decorator replaces the `for` loop. Once you call a function decorated with `@q.worker`, the code never returns unless there is an unhandled exception.

Exercise. Write a function, `echo(item)`, to print an item to the screen, and use the `q.worker` decorator to turn it into a consumer. Call your echo function in one terminal and in a separate terminal, send messages to the redis queue. Verify that the message items are printed to the screen in the first terminal.

In practice, we will use the `@q.worker` in a Python source file like so –

```
# A simple example of Python source file, worker.py
q = HotQueue("some_queue", host="<Redis_IP>", port=6379, db=1)

@q.worker
def do_work(item):
    # do something with item...

do_work()
```

Assuming the file above was saved as `worker.py`, calling `python worker.py` from the shell would result in a non-terminating program that "processed" the items in the `"some_queue"` queue using the `do_work(item)` function. The only thing that would cause our worker to stop is an unhandled exception.

10.2.3 Concurrency in the Jobs API

Recall that our big-picture goal is to add a Jobs endpoint to our Flask system that can process long-running tasks. We will implement our Jobs API with concurrency in mind. The goals will be:

- Enable analysis jobs that take longer to run than the request/response cycle (typically, a few seconds or less).
- Deploy multiple "worker" processes to enable more throughput of jobs.

The overall architecture will thus be:

a) Save the request in a database and respond to the user that the analysis will eventually be run.

b) Give the user a unique identifier with which they can check the status of their job and fetch the results when they are ready,

c) Queue the job to run so that a worker can pick it up and run it.

d) Build the worker to actually work the job.

Parts a), b) and c) are the tasks of the Flask API, while part d) will be a worker, running as a separate pod/container, that is waiting for new items in the Redis queue.

10.2.4 Code Organization

As software systems get larger, it is very important to keep code organized so that finding the functions, classes, etc. responsible for different behaviors is as easy as possible. To some extent, this is technology-specific, as different languages, frameworks, etc., have different rules and conventions about code organization. We'll focus on Python, since that is what we are using.

The basic unit of code organization in Python is called a "module". This is just a Python source file (ends in a `.py` extension) with variables, functions, classes, etc., defined in it. We've already used a number of modules, including modules that are part of the Python standard library (e.g. `json`) and modules that are part of third-party libraries (e.g., `redis`).

The following should be kept in mind when designing the modules of a larger system:

- Modules should be focused, with specific tasks or functionality in mind, and their names (preferably, short) should match their focus.

- Modules are also the most typical entry-point for the Python interpreter itself, (e.g., `python some_module.py`).
- Accessing code from external modules is accomplished through the `import` statement.
- Circular imports will cause errors - if module A imports an object from module B, module B cannot import from module A.

Examples. The Python standard library is a good source of examples of module design. You can browse the standard library for Python 3.9 here.

- We see the Python standard library has modules focused on a variety of computing tasks; for example, for working with different data types, such as the `datetime` module and the `array` module. The descriptions are succinct:
  - *The datetime module supplies classes for manipulating dates and times.*
  - *This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers*
- For working with various file formats: e.g., `csv`, `configparser`
- For working with concurrency: `threading`, `multiprocessing`, etc.

With this in mind, a first approach might be to break up our system into two modules:

- `api.py` - this module contains the flask web server.
- `worker.py` - this module contains the code to execute jobs.

However, both the API server and the workers will need to interact with the database and the queue:

- The API will create new jobs in the database, put new jobs onto the queue, and retrieve the status of jobs (and probably the output products of the job).
- The worker will pull jobs off the queue, retrieve jobs from the database, and update them.

This suggests a different structure:

- `api.py` - this module contains the flask web server.
- `jobs.py` - this module contains core functionality for working with jobs in Redis (and on the queue).
- `worker.py` - this module contains the code to execute jobs.

Common code for working with `redis`/`hotqueue` can go in the `jobs.py` module and be imported in both `api.py` and `worker.py`.

Note: High-quality modular design is a crucial aspect of building good software. It requires significant thought and experience to do correctly, and when done poorly it can have dire consequences. In the best case, poor module design can make the software difficult to maintain/upgrade; in the worst case, it can prevent it from running correctly at all.

10.2.5 Private vs Public Objects

As software projects grow, the notion of public and private access points (functions, variables, etc.) becomes an increasingly important part of code organization.

- Private objects should only be used within the module they are defined. If a developer needs to change the implementation of a private object, she only needs to make sure the changes work within the existing module.
- Public objects can be used by external modules. Changes to public objects need more careful analysis to understand the impact across the system.

Like the layout of code itself, this topic is technology-specific. In this class, we will take a simplified approach based on our use of Python. Remember, this is a simplification to illustrate the basic concepts - in practice, more advanced/robust approaches are used.

- We will name private objects starting with a single underscore (_) character.
- If an object does not start with an underscore, it should be considered public.

We can see public and private objects in use within the standard library as well. If we open up the source code for the `datetime` module, which can be found on GitHub we see a mix of public and private objects and methods.

- Private objects are listed first.
- Public objects start on line 473 with the `timedelta` class.

Exercise. Create three files, `api.py`, `worker.py` and `jobs.py` in your local repository, and update them by working through the following example.

Here are some function and variable definitions, some of which have incomplete implementations and/or have invalid syntax.

To begin, place them in the appropriate files. Also, determine if they should be public or private.

```
def _generate_jid():
    return str(uuid.uuid4())

app = Flask(__name__)

def _generate_job_key(jid):
    return 'job.{}'.format(jid)

q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)

def _instantiate_job(jid, status, start, end):
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
        }
    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')
    }

@app.route('/jobs', methods=['POST'])
def jobs_api():
    try:
        job = request.get_json(force=True)
```

(continues on next page)

(continued from previous page)

```
    except Exception as e:
        return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}.'.
↪format(e)})
    return json.dumps(jobs.add_job(job['start'], job['end']))

def _save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hmset(.......)

def _queue_job(jid):
    """Add a job to the redis queue."""
    ....

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

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = _instantiate_job(jid, status, start, end)
    _save_job(......)
    _queue_job(......)
    return job_dict

@<...>   # fill in
def execute_job(jid):
    # fill in ...

rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)

def update_job_status(jid, status):
    """Update the status of job with job id `jid` to status `status`."""
    jid, status, start, end = rd.hmget(generate_job_key(jid), 'id', 'status', 'start',
↪ 'end')
    job = _instantiate_job(jid, status, start, end)
    if job:
        job['status'] = status
        _save_job(_generate_job_key(jid), job)
    else:
        raise Exception()
```

Solution. We start by recognizing that `app = Flask(__name__)` is the instantiation of a Flask app, the `@app.route` is a flask decorator for defining an endpoint in the API, and the `app.run` line is used to launch the flask server, so we add those both in the `api.py` file:

```
# api.py

  app = Flask(__name__)

  @app.route('/jobs', methods=['POST'])
  def jobs_api():
      try:
          job = request.get_json(force=True)
      except Exception as e:
          return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}.'.
↪format(e)})
```

(continues on next page)

(continued from previous page)

```
    return json.dumps(jobs.add_job(job['start'], job['end']))

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

We also recognize several functions are private by the leading _ in the name. They are:

- `_generate_jid`
- `_generate_job_key`
- `_instantiate_job`
- `_save_job`
- `_queue_job`

These all have to do with jobs and are used (either directly or indirectly) by the `add_job` function. One more hint is that the `jobs_api()` function, which we just put in `api.py`, actually references `jobs.add_job`, so we can put these in the `jobs.py` file:

```
# jobs.py
def _generate_jid():
    return str(uuid.uuid4())

def _generate_job_key(jid):
    return 'job.{}'.format(jid)

def _instantiate_job(jid, status, start, end):
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
        }
    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')
    }

def _save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hmset(.......)

def _queue_job(jid):
    """Add a job to the redis queue."""
    ....

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = _instantiate_job(jid, status, start, end)
    _save_job(......)
    _queue_job(......)
    return job_dict
```

That leaves the definition of the `q = HotQueue(..)`, `rd = StrictRedis(..)`, `update_job_status()` and `execute_job()`.

- We know `worker.py` is responsible for actually executing the job, so `execute_job` should go there.
- The `update_job_status()` is a jobs-related task, so it goes in the `jobs.py` file – it also makes a call to `_instantiate_job` which is already in `jobs.py`.
- The jobs.py file definitely needs access to the `rd` object so that goes there.
- Lastly, the `q` will be needed by both `jobs.py` and `worker.py`, but `worker.py` is already importing from `jobs`, so we better put it in `jobs.py` as well.

Therefore, the final solution is:

```
# api.py

  app = Flask(__name__)

  @app.route('/jobs', methods=['POST'])
  def jobs_api():
      try:
          job = request.get_json(force=True)
      except Exception as e:
          return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}.'.
↪format(e)})
      return json.dumps(jobs.add_job(job['start'], job['end']))

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

```
# jobs.py
  q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)
  rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)

  def _generate_jid():
      return str(uuid.uuid4())

  def _generate_job_key(jid):
      return 'job.{}'.format(jid)

  def _instantiate_job(jid, status, start, end):
      if type(jid) == str:
          return {'id': jid,
                  'status': status,
                  'start': start,
                  'end': end
          }
      return {'id': jid.decode('utf-8'),
              'status': status.decode('utf-8'),
              'start': start.decode('utf-8'),
              'end': end.decode('utf-8')
      }

  def _save_job(job_key, job_dict):
      """Save a job object in the Redis database."""
      rd.hmset(.......)

  def _queue_job(jid):
      """Add a job to the redis queue."""
      ....
```

(continues on next page)

(continued from previous page)

```
  def add_job(start, end, status="submitted"):
      """Add a job to the redis queue."""
      jid = _generate_jid()
      job_dict = _instantiate_job(jid, status, start, end)
      _save_job(......)
      _queue_job(......)
      return job_dict

  def update_job_status(jid, status):
      """Update the status of job with job id `jid` to status `status`."""
      jid, status, start, end = rd.hmget(generate_job_key(jid), 'id', 'status', 'start
↪', 'end')
      job = _instantiate_job(jid, status, start, end)
      if job:
          job['status'] = status
          _save_job(_generate_job_key(jid), job)
      else:
          raise Exception()
```

```
# worker.py
  @<...>   # fill in
  def execute_job(jid):
      # fill in ...
```

Exercise. After placing the functions in the correct files, add the necessary `import` statements.

Solution. Let's start with `api.py`. We know we need to import the `Flask` class to create the `app` object and to use the flask `request` object. We also use the `json` package from the standard library. Finally, we are using our own `jobs` module.

```
# api.py
import json
from flask import Flask, request
import jobs

app = Flask(__name__)

@app.route('/jobs', methods=['POST'])
def jobs_api():
    try:
        job = request.get_json(force=True)
    except Exception as e:
        return True, json.dumps({'status': "Error", 'message': 'Invalid JSON: {}.'.
↪format(e)})
    return json.dumps(jobs.add_job(job['start'], job['end']))

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

For `jobs.py`, there is nothing from our own code to import (which is good since the other modules will be importing from it, but we do need to import the `StrictRedis` and `HotQueue` classes. Also, don't forget the use of the `uuid` module from the standard lib! So, `jobs.py` becomes:

```
# jobs.py
import uuid
from hotqueue import HotQueue
```

(continues on next page)

(continued from previous page)

```
from redis import StrictRedis

q = HotQueue("queue", host='172.17.0.1', port=6379, db=1)
rd = StrictRedis(host='172.17.0.1', port=6379, db=0)

def _generate_jid():
    return str(uuid.uuid4())

def _generate_job_key(jid):
    return 'job.{}'.format(jid)

def _instantiate_job(jid, status, start, end):
    if type(jid) == str:
        return {'id': jid,
                'status': status,
                'start': start,
                'end': end
        }
    return {'id': jid.decode('utf-8'),
            'status': status.decode('utf-8'),
            'start': start.decode('utf-8'),
            'end': end.decode('utf-8')
    }

def _save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hmset(.......)

def _queue_job(jid):
    """Add a job to the redis queue."""
    ....

def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = _instantiate_job(jid, status, start, end)
    _save_job(......)
    _queue_job(......)
    return job_dict

def update_job_status(jid, status):
    """Update the status of job with job id `jid` to status `status`."""
    jid, status, start, end = rd.hmget(generate_job_key(jid), 'id', 'status', 'start',
↪ 'end')
    job = _instantiate_job(jid, status, start, end)
    if job:
        job['status'] = status
        _save_job(_generate_job_key(jid), job)
    else:
        raise Exception()
```

Finally, on the surface it doesn't appear that the worker needs to import anything, but we know it needs the `q` object to get items. It's hidden by the missing decorator. Let's go ahead and import it:

```
# worker.py
from jobs import q
```

(continues on next page)

(continued from previous page)

```
@<...>   # fill in
def execute_job(jid):
    # fill in ...
```

Exercise. Write code to finish the implementations for `_save_job` and `_queue_job`.

Solution. The `_save_job` function should save the job to the database, while the `_queue_job` function should put it on the queue. We know how to write those:

```
def _save_job(job_key, job_dict):
    """Save a job object in the Redis database."""
    rd.hmset(job_key, job_dict)

def _queue_job(jid):
    """Add a job to the redis queue."""
    q.put(jid)
```

Exercise. Fix the calls to `_save_job` and `execute_job` within the `add_job` function. *Solution.* The issue in each of these are the missing parameters. The `_save_job` takes `job_key, job_dict`, so we just need to pass those in. Similarly, `_queue_job` takes `jid`, so we pass that in. The `add_job` function thus becomes:

```
def add_job(start, end, status="submitted"):
    """Add a job to the redis queue."""
    jid = _generate_jid()
    job_dict = _instantiate_job(jid, status, start, end)
    # update call to save_job:
    save_job(_generate_job_key(jid), job_dict)
    # update call to queue_job:
    queue_job(jid)
    return job_dict
```

Exercise. Finish the `execute_job` function. This function needs a decorator (which one?) and it needs a function body.

The function body needs to:

- update the status at the start (to something like "in progress").
- update the status when finished (to something like "complete").

For the body, we will use the following (incomplete) simplification:

```
update_job_status(jid, .....)
# todo -- replace with real job.
time.sleep(15)
update_job_status(jid, .....)
```

Solution. As discussed before, we saw in class we can use the `q.worker` decorator to turn the worker into a consumer.

As for `execute_job` itself, we are given the body, we just need to fix the calls to the `update_job_status()` function. The first call puts the job "in progress" while the second sets it to "complete". So the function becomes:

```
@<...>   # fill in
def execute_job(jid):
    update_job_status(jid, "in progress")
    time.sleep(15)
    update_job_status(jid, "complete")
```

Note that we are using the `update_job_status` function from `jobs.py` now, so we need to import it. The final `worker.py` is thus:

```
from jobs import q, update_job_status

@q.worker
def execute_job(jid):
    jobs.update_job_status(jid, 'in progress')
    time.sleep(15)
    jobs.update_job_status(jid, 'complete')
```

CHAPTER

ELEVEN

WEEK 12: ASYNCHRONOUS PROGRAMMING II

This week we complete our treatment of asynchronous programming.

11.1 Deploying to k8s

In this lecture, we will bring everything together and deploy an updated version of our Flask API system to k8s that includes a Jobs endpoint and a worker deployment. But first, we need to finish the worker.

11.1.1 Daemonizing the Worker

In a Unix-like operating system, a *daemon* is a type of program that runs unobtrusively in the background, rather than under the direct control of a user. The daemon waits to be activated by an occurrence of a specific event or condition.

In summary: A daemon is a long-running background process that answers requests or responds to events.

Recall the high-level architecture of our Jobs API:

- Our Flask API will play the role of producer.
- One or more "worker" programs will play the role of consumer.
- Workers will receive messages about new jobs to execute and performing the analysis steps.
- Workers will oversee the execution of the analysis steps and update the database with the results.

Therefore, our worker program is an example of a daemon that will simply run in the background, waiting for new messages to arrive and executing the corresponding jobs.

We have actually already seen how to turn our Python code into a worker daemon. Let us recall that here:

- We create a new file, `worker.py`, where we put all code related to processing a job.
- The `worker.py` will import a queue object from a `jobs.py` module
- The `worker.py` file includes a function that can take a message from the queue and start processing a job.
- The worker will use the queue object's `worker` decorator to turn this function into a consumer.
- By adding a call to the function at the bottom of `worker.py`, the worker can be run as a daemon.

Here is a skeleton of the worker.py module –

```
# worker.py skeleton
from jobs import q

@q.worker
```

(continues on next page)

(continued from previous page)

```
def do_work(item):
    # do something with item...

do_work()
```

To execute our worker, we simply issue the command `python worker.py` from the command line. Let's step through what happens, just to make sure this is clear.

1. When `python worker.py` is called from the command line, the python interpreter reads each line of the `worker.py` file and executes any statements it finds in order, from top to bottom.
2. The first line it encounters is the import statement. This imports the definition of `q` from the `jobs.py` file (not included above).
3. Next it hits the decorator and the definition of the function, `do_work(item)`. It checks the syntax of this definition.
4. Finally, it executes the `do_work()` function at the bottom. Since this function is decorated with the `q.worker` decorator, it runs indefinitely, consuming messages from the Redis `q` queue.

11.1.2 Containerizing the Worker

There are multiple ways to containerize the worker, but the simplest approach is to add the `worker.py` code to the same image with the flask API code, and use different commands when running the web server vs running the worker.

For example, the Dockerfile could look like:

```
# Image: jstubbs/animals-service
FROM python:3.9

ADD requirements.txt /requirements.txt
RUN pip install -r requirements.txt
COPY source /app
WORKDIR /app

ENTRYPOINT ["python"]
COMMAND ["app.py"]
```

When running the flask application, the entrypoint and command are already correct. For running the worker, we simply update the command to be "worker.py" instead of "app.py".

Exercise. Update your Dockerfile to include an entrypoint and a command that can be used for running both the flask web application and the worker program. Build the new version of your image and push it to Docker Hub.

11.1.3 Deploying to k8s

We're now ready to deploy our complete system to k8s. You should already have deployments and services in k8s for the Flask API and the Redis database, and you should also already have a PVC for Redis to persist state to a volume.

What's left is to add a deployment for the worker pods. Do we need to add a service or PVC for the workers? Why or why not?

Exercise. Create a deployment for your worker pods. Put 2 replicas and be sure to set the command correctly. See above. A deployment skeleton is included below for you to use if you like. Think through the values of each section; some properties/stanzas may not be needed for the worker.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <...>
  labels:
    app: <...>
spec:
  replicas: <...>
  selector:
    matchLabels:
      app: <...>
  template:
    metadata:
      labels:
        app: <...>
    spec:
      containers:
        - name: <...>
          imagePullPolicy: Always
          image: <...>
          command: <...>
          env:
          - <...>
          ports:
          - <...>
```

11.1.4 Code Repository

It is good to keep your code and deployment files organized in a single repository. Consider using a layout similar to the following:

```
deploy/
  api/
    deployment.yml
    service.yml
  db/
    deployment.yml
    pvc.yml
    service.yml
  worker/
    deployment.yml
Dockerfile
source/
  api.py
  jobs.py
  worker.py
```

CHAPTER

TWELVE

WEEK 13: CONTINUOUS INTEGRATION, INTEGRATION TESTING

This final unit of the class will work through a sample final project, from conceptualization, to design, development, testing, and finally deployment into production. Along the way, we will cover concepts in continuous integration and in integration testing.

12.1 Conceptualization

This demo final project for COE 332 is called **PSSP API**. It is a REST API interface to a scientific code base. The scientific code used here is a protein secondary structure prediction (PSSP) tool called Predict_Property.

12.1.1 Background

Protein secondary structure, or the local folded structures that form within a protein, can be predicted somewhat successfully from the primary amino acid sequence. Researchers have developed tools that take amino acid sequence as input (e.g. 'AAAAAAA'), and return the likelihood of different folded structures occurring at each position.

Many of these tools are Linux command line tools. It would be useful and interesting to have an intuitive REST API for calling one of these tools, so that users can perform these prediction calculations without necessarily having command line experience or without installing the tool themselves. Further, this is a first step toward encapsulating the function of the scientific code base into a web interface.

12.1.2 Scope

The scope of this project is narrow. It should be designed to expect only one kind of input: protein primary sequences as a string of letters. It performs the same standard analysis using the *Predict_Property* command line tools each time. The expected results returned always follow the same 8-line format shown below (# annotations on each line not included with the result):

```
> Header Info      #-> header / metadata including job id
ASDFASDGFAGASG     #-> user input sequence with invalid amino acid shown as 'X'.
HHHHEEECCCCCHH     #-> 3-class secondary structure (SS3) prediction.
HHGGEEELLSSTHH     #-> 8-class secondary structure (SS8) prediction.
EEMMEEBBEEEBBM     #-> 3-state solvent accessibility (ACC) prediction.
*****.....***      #-> disorder (DISO) prediction, with disorder residue shown as '*'.
_____HHHHH____     #-> 2-class transmembrane topology (TM2) prediction.
UU___HHHHH____     #-> 8-class transmembrane topology (TM8) prediction.
```

Here are a few tables for interpreting the results:

SS3		SS8	
Key	Value	Key	Value
H	a-helix	H	a-helix
E	b-sheet	G	3-helix
C	coil	I	5-helix
.	.	E	b-strand
.	.	B	b-bridge
.	.	T	turn
.	.	S	bend
.	.	L	loop

ACC		DISO	
Key	Value	Key	Value
B	buried	*	disordered
M	medium	_	not disordered
E	exposed	.	.

TM2		TM8	
Key	Value	Key	Value
H	transmembrane	H	transmem helix
_	not transmembrane	E	transmem strand
.	.	C	transmem coil
.	.	I	membrane-inside
.	.	L	membrane-loop
.	.	F	interfacial helix
.	.	X	unknown localizations
.	.	_	not transmembrane

12.1.3 User Interface

With this API, users should have access to a number of curl routes to GET information about the service, and about past jobs that have been run. E.g.:

```
curl localhost:5041/             # general info
curl localhost:5041/run          # get instructions to submit a job
curl localhost:5041/jobs         # get past jobs
curl localhost:5041/jobs/JOBID   # get results for JOBID
```

Users should also be able to POST protein primary sequences (e.g 'AAAAAAA') to a defined route, and in return they will receive a Job ID.

```
curl -X POST -d "seq=AAAAA" localhost:5041/run
```

12.1.4 Technologies / Architecture

There will be two "environments" used to develop, test, and deploy this API. They will be referred to as:

- **The Development Environment** refers to the class ISP server. This env will be used to develop new features / routes in the source code. Containers will be built and deployed using docker and docker-compose commands. Services will be tested by directly connecting to the containers.
- **The Deployment Environment** refers to the Kubernetes cluster. This env will host both a testing (also called "staging") and production deployment of the full API. No code edits will take place in this environment. It will exclusively pull pre-built / tagged containers from Docker Hub for the runtime.

Note: Our deployment environment is a local Kubernetes Cluster, but it could just as easily have been AWS, Azure, Google Cloud, etc.

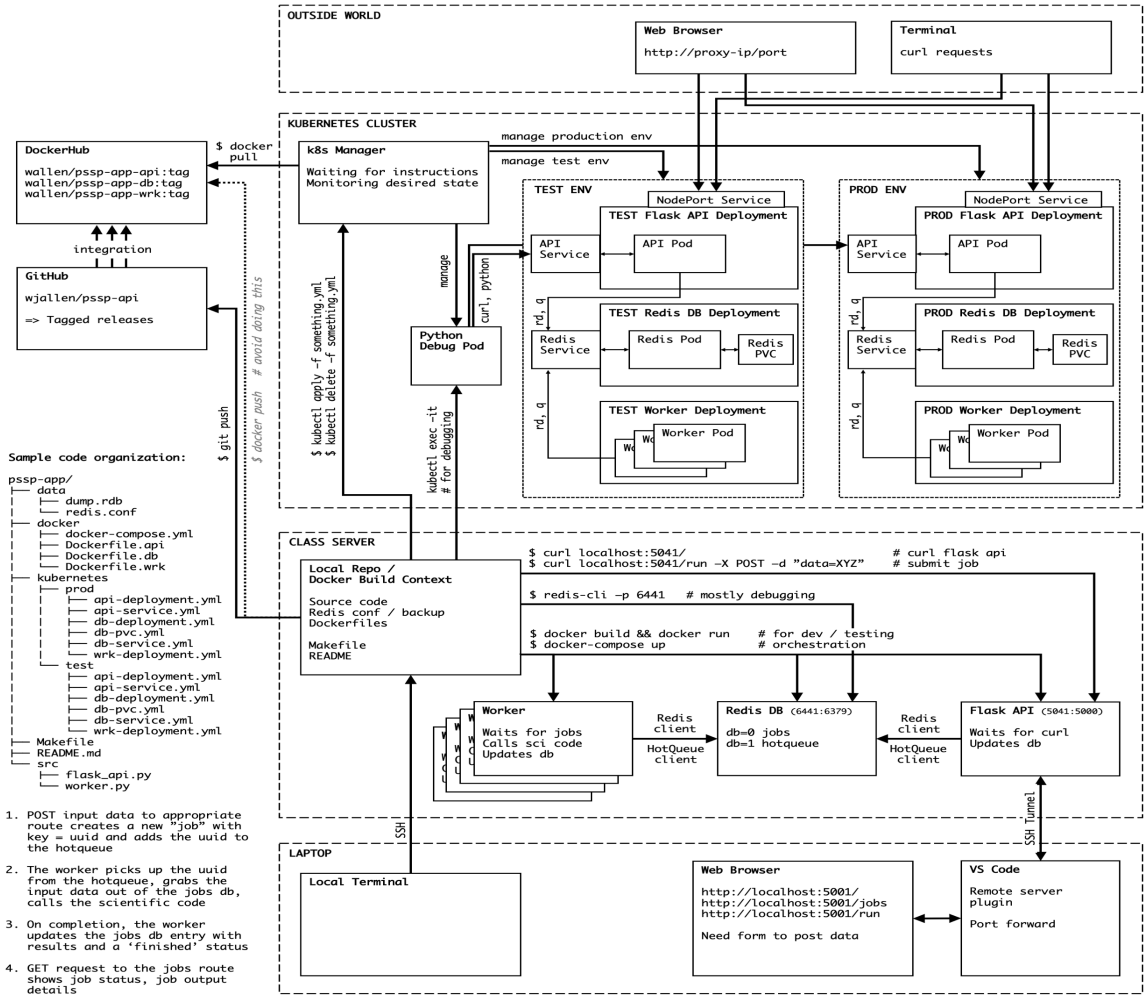


Fig. 1: Design diagram

The different components of this environment will be described in the following pages.

12.2 Development Environment

Development of this API will be performed on the ISP server. The containers and endpoints developed here will be ephemeral, only lasting long enough to test and debug new features. They should never be seen by end users. There will be three containerized components:

1. A Flask API front end for submitting / accessing jobs
2. A Redis database for storing job and queue information
3. A worker back end which runs the PSSP code

12.2.1 File Organization

An example file organization scheme for developing this API may look like:

```
pssp-api/
├── data
│   ├── dump.rdb
│   └── redis.conf
├── docker
│   ├── docker-compose.yml
│   ├── Dockerfile.api
│   ├── Dockerfile.db
│   └── Dockerfile.wrk
├── README.md
└── src
    ├── flask_api.py
    └── worker.py
```

In this example, the *data/* subfolder is mounted inside the Redis container. The *redis.conf* configuration file is useful to have to customize how the database behaves, and all the data is captured in regular intervals as *dump.rdb* for easy backups.

The *docker/* subfolder contains a Dockerfile for each service, plus a *docker-compose.yml* for orchestrating all services at once.

The *src/* folder contains the source Python scripts that are injected into the API and worker containers. This is where the majority of the development will occur.

Tip: It is a very, very good idea to put this whole folder under version control.

12.2.2 Docker

We previously talked at great length about why it is a good idea to containerize an app / service that you develop. One of the main reasons was for portability to other machines. All the development / testing done in this development environment will directly translate to our deployment environment (Kubernetes), as we will see in the next module.

The development cycle for each of the three containerized components generally follows the form:

1. Edit some source code (e.g. add a new Flask route)
2. Delete any running container with the old source code
3. Re-build the image with `docker build`
4. Start up a new container with `docker run`
5. Test the new code / functionality
6. Repeat

This 6-step cycle is great for iterating on each of the three containers independently, or all at once. However, watch out for potential error sources. For example if you take down the Redis container, a worker container that is in the middle of watching the queue may also go down and will need to be restarted (once a new Redis container is up).

12.2.3 Makefile

Makefiles can be a useful automation tool for testing your services. Many code projects use Makefiles to help with the compile and install process (e.g. `make && make install`). Here, we will set up a Makefile to help with the 6-step cycle above. Using certain keywords (called "targets") we will create shortcuts to cleaning up running containers, re-building docker images, running new containers, and deploying it all with docker-compose.

Targets are listed in a file called `Makefile` in this format:

```
target: prerequisite(s)
        recipe
```

Targets are short keywords, and recipes are shell commands. For example, a simple target might look like:

```
ps-me:
        docker ps -a | grep wallen
```

Put this text in a file called `Makefile` in your current directory, and then you simply need to type:

```
[isp02]$ make ps-me
```

And that will list all the docker containers with the username 'wallen' either in the image name or the container name. Makefiles can be further abstracted with variables to make them a little bit more flexible. Consider the following Makefile:

```
NAME ?= wallen

all: ps-me im-me

im-me:
        docker images | grep ${NAME}

ps-me:
        docker ps -a | grep ${NAME}
```

Here we have added a variable `NAME` at the top so we can easily customize the targets below. We have also added two new targets: `im-me` which lists images, and `all` which does not contain any recipes, but does contain two prerequisites - the other two targets. So these two are equivalent:

```
# make all targets
[isp02]$ make all

# or make them one-by-one
[isp02]$ make ps-me
[isp02]$ make im-me

# Try this out:
[isp02]$ NAME="redis" make all
```

EXERCISE

Write a Makefile that, at a minimum:

1. Builds all necessary images for your app from Dockerfile(s)
2. Starts up new containers / services
3. Removes running containers in your namespace (be careful!)

12.2.4 Docker-Compose

Although it is not strictly necessary, it might also be useful to write Makefile targets to run a `docker-compose` deployment of all of your services as a unit. This behavior more closely mimics what it will be like to put services up in your Kubernetes deployment environment. Be careful, however, about the order in which docker-compose starts services. If the Redis DB service is not ready, your worker service(s) may exit immediately with an error like 'Can not connect to database'.

12.3 Deployment Environment

Our deployment environment for this API will be the class Kuberentes cluster. It could just as easily be AWS, or Azure, or Google Cloud, or another Kuberenets cluster. Remember if you containerize everything, it becomes extremely portable. In contrast to our development environment, the Kubernetes deployment is meant to be long-lasting, always available, and consumable by the public. We will have *test* and *prod* deployments, so that new changes can be seen by developers in the *test* deployment environment (sometimes also called "staging") before finally making their way to the *prod* (production) deployment environment.

12.3.1 File Organization

To support the deployment environment, our file organization grows to the following:

```
pssp-api/
├── data
│   ├── dump.rdb
│   └── redis.conf
├── docker
│   ├── docker-compose.yml
│   ├── Dockerfile.api
```

(continues on next page)

(continued from previous page)

```
    ├── Dockerfile.db
    └── Dockerfile.wrk
├── kubernetes
    ├── prod
        ├── api-deployment.yml
        ├── api-service.yml
        ├── db-deployment.yml
        ├── db-pvc.yml
        ├── db-service.yml
        └── wrk-deployment.yml
    └── test
        ├── api-deployment.yml
        ├── api-service.yml
        ├── db-deployment.yml
        ├── db-pvc.yml
        ├── db-service.yml
        └── wrk-deployment.yml
├── Makefile
├── README.md
└── src
    ├── flask_api.py
    └── worker.py
```

Here you will find 12 new yaml files with somwhat descriptive names. Six are organized into a 'test' directory, and six are organized into a 'prod' directory.

These yaml files closely follow the naming convention and content we have seen in previous lectures.

12.3.2 Testing

The purpose of this testing / staging environment is to see the entire API exactly as it appears in production before actually putting new code changes into production.

Generally the process to get code into testing follows these steps:

1. Develop / test code in the development environment (ISP) as described in the previous module
2. Push code to GitHub and tag it with an appropriate version number (avoid using "latest")
3. Push images to Docker Hub - Kubernetes needs to pull from here. Make sure the Docker image tag matches the GitHub tag so you always know what exact version of code is running.
4. Edit the appropriate testing deployment(s) with the new tags and apply the changes. Pods running within a deployment under the old tag number should be automatically terminated.

The yaml files above can be applied one by one, or the entire directory at a time like the following:

```
[isp02]$ kubectl apply -f kubernetes/test/
```

Kuberenets will apply all the files found in the test folder. Be careful, however, about the order in which things are applied. For example, the Redis DB deployment needs the PVC to exist in order to deploy successfully. But, Kubernetes is usually pretty smart about this kind of thing, so it should keep retrying all deployments, services, and pvcs until everything is happy and connected.

Once deployed, you should rigorously test all services using the python debug pod and, if applicable, the NodePort Service connection to the outside world. We will see more on automating integration tests later in this unit.

12.3.3 Production

If everything with the test / staging deployment looks good and passes tests, follow the same steps for your production environment. Kubernetes is fast at stopping / starting containers, and the services should provide pretty seemless access to the underlying API. If larger-scale changes are needed and significant downtime is anticipated, it would be a good idea to post an outage notice to users.

12.4 Continuous Integration

The primary goal of Continuous Integration (CI) is to enable multiple developers to work on the same code base while ensuring the quality of the final product.

This involves the following challenges:

- No one person has complete knowledge of the entire system.
- Multiple changes can be happening at the same time. Even if the changes are made in different components, it is possible for something to break when they are integrated.

12.4.1 Enabling Large Teams to Work on a System Simultaneously

An "integration server" (or "build server") is a dedicated server (or VM) that prepares software for release. The server automates common tasks, including:

- Building software binaries from source code (for compiled languages)
- Running tests
- Creating images, installers, or other artifacts
- Deploying/installing the software

We are ultimately aiming for the following "Continuous Integration" work flow or process; this mirrors the process used by a number of teams working on "large" software systems, both in academia and industry:

- Developers (i.e., you) check out code onto a machine where they will do their work. This could be a VM somewhere or their local laptop.
- They make changes to the code to add a feature or fix a bug.
- Once their work is done they add any additional tests as needed and then run all unit tests "locally" (i.e., on the same machine).
- Assuming the tests pass, the develop commits their changes and pushes to the remote origin (in this case, GitHub).
- A pre-established build server gets a message from the origin that a new commit was pushed.
- The build server:
  - Checks out the latest version
  - Executes any build steps to create the software
  - Runs unit tests
  - Starts an instance of the system
  - Runs integration tests
  - Deploys the software to a staging environment

If any one of the steps above fails, the process stops. In such a situation, the code defect should be addressed as soon as possible.

12.4.2 Popular Automated CI Services

Jenkins is one of the most popular free open-source CI services. It is server-based, and it requires a web server to operate on.

- Local application
- Completely free
- Deep workflow customization
- Intuitive web interface management
- Can be distributed across multiple machines / VMs
- Rich in features and plugins
- Easy installation thanks to the pre-installed OS X, Unix and Windows packages
- A well-established product with an excellent reputation

TravisCI is another CI service with limited features in the free tier, and a comprehensive paid tier. It is a cloud-hosted service, so there is no need for you to host your own server.

- Quick setup
- Live build views
- Pull request support
- Multiple languages and platforms support
- Pre-installed database services
- Auto deployments on passing builds
- Parallel testing (paid tier)
- Scaling capacity on demand (paid tier)
- Clean VMs for every build
- Mac, Linux, and iOS support
- Connect with Github, Bitbucket and more

GitHub Actions is a relatively new CI service used to automate, customize, and execute software development workflows right in your GitHbu repository.

- One interface for both your source code repositories and your CI/CD pipelines
- Catalog of available Actions you can utilize without reinventing the wheel
- Includes macOS builds as part of free tier (macOS builds require 10 credits per build minute however compared to 1 credit per minute for linux)
- Free tier credits are renew/reset each month
- It is a new platform, so not as many features as some of the others

12.4.3 GitHub-Docker Hub Integration

A quick-and-easy way to implement some parts of CI for our app is to use a handy GitHub-Docker Hub Integration.

Rather than commit to GitHub AND push to Docker Hub each time you want to release a new version of code, you can set up an integration between the two services that automates it. The key benefit is you only have to commit to one place (GitHub), and you know the image available on Docker Hub is always in sync.

To set up the integration, navigate to an existing Docker repository that you own in a web browser, which should be at an address similar to:

https://hub.docker.com/repository/docker/YOUR-DOCKER-USERNAME/pssp-app-api

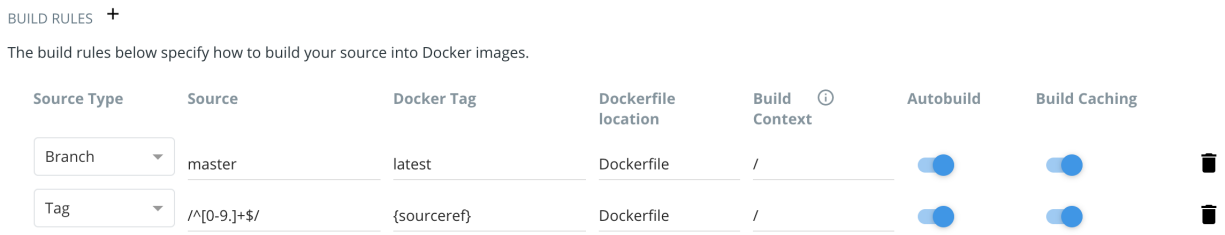
---

Tip: You can click `+ New Repository` if it doesn't exist yet.

Click on Builds => Link to GitHub. (If this is your first time connecting a Docker repo to a GitHub repo, you will need to set it up. Press the 'Connect' link to the right of 'GitHub'. If you are already signed in to both Docker and GitHub in the same browser, it takes about 15 seconds to set up).

Once you reach the Build Configurations screen, you will select your GitHub username and repository named, e.g., `pssp-api`.

Leaving all the defaults selected will cause this Docker image to rebuild every time you push code to the master branch of your GitHub repo. For this example, set the build to to trigger whenever a new release is tagged:



Click 'Save and Build' and check the 'Timeline' tab on Docker Hub to see if it is working as expected.

EXERCISE

In your own GitHub account, fork this repository:

https://github.com/wjallen/pssp-api

Create the same GitHub - Docker Hub integration shown above, write a new feature and tag a new release as described below to trigger a build.

12.4.4 Commit to GitHub

To trigger the build in a real-world scenario, make some changes to your source code, push your modified code to GitHub and tag the release as `X.Y.Z` (whatever new tag is appropriate) to trigger another automated build:

```
$ git add *
$ git commit -m "added a new route to delete jobs"
$ git push
$ git tag -a 0.1.4 -m "release version 0.1.4"
$ git push origin 0.1.4
```

By default, the git push command does not transfer tags, so we are explicitly telling git to push the tag we created (0.1.4) to the remote (origin).

Now, check the online GitHub repo to make sure your change / tag is there, and check the Docker Hub repo to see if your image is automatically rebuilding.

12.4.5 Deploy to Kubernetes

The final step in our example is to update the image tag in our deployment yaml files in both test and prod, and apply them all. Apply to test (staging) first as one final check that things are working as expected. Then, deploy to prod. Because the old containers are Running right up until the moment the new containers are deployed, there is virtually no disruption in service.

Note: Some CI / CD services can even handle the deployment to Kubernetes following Docker image builds and passing tests.

12.5 Integration Testing

Unlike unit tests, integration tests exercise multiple components, functions, or units of a software system at once. Some properties of integration tests include:

- Each test targets a higher-level capability, requirement or behavior of the system, and exercises multiple components of the system working together.
- Broader scope means fewer tests are required to cover the entire application/system.
- A given test failure provides more limited information as to the root cause.

It's worth pointing out that our definition of integration test leaves some ambiguity. You will also see the term "functional tests" used for tests the exercise entire aspects of a software system.

12.5.1 Challenges When Writing Integration Tests

Integration tests against large, distributed systems with lots of components that interact face some challenges.

- We want to keep tests independent so that a single test can be run without its result depending on other tests.
- Most interesting applications change "state" in some way over time; e.g., files are saved/updated, database records are written, queue systems updated. In order to properly test the system, specific state must be established before and after a test (for example, inserting a record into a database before testing the "update" function).
- Some components have external interactions, such as an email server, a component that makes an update in an external system (e.g. GitHub) etc. A decision has to be made about whether or not this functionality will be validated in the test and if so, how.

12.5.2 Initial Integration Tests for Our Flask API

For our first set of integration tests, we'll use the following strategy:

- Start the Flask API, Redis DB, and Worker services
- Use `pytest` and `requests` to make requests directly to the running API server
- Check various aspects of the response; each check can be done with a simple assert statement, just like for unit tests

12.5.3 A Simple pytest Example

Similar to unittests, we will use *assert* statements to check that some input data or command returns the expected result. A simple example of using pytest might look like:

```
1 import pytest, requests
2
3 def test_flask():
4     response = requests.get('http://localhost:5000/route')
5     assert response.status_code == 200
```

This small test just checks to make sure curling the route (with the Python requests library) returns a successful status code, `200`.

As we have seen before, test scripts should be named strategically and organized into a subdirectory similar to:

```
pssp-app/
├── data
├── docker
├── kubernetes
│   ├── prod
│   └── test
├── Makefile
├── README.md
├── src
│   ├── flask_api.py
│   └── worker.py
└── test
    └── test_flask.py
```

Run the test simply by typing this in the top (`pssp-app/`) directory:

```
[isp02]$ pytest
========================= test session starts ==========================
platform linux -- Python 3.6.8, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /home/wallen/coe-332/pssp-app
collected 1 item

test/test_flask.py .                                             [100%]

========================== 1 passed in 0.17s ===========================
```

Tip: You may have to `pip3 install --user pytest` first.

EXERCISE

Continue working in the test file, `test_flask.py`, and write a new functional test that use the `requests` library to make a `GET` request to the `/jobs` endpoint and check the response for, e.g.:

- The response returns a 200 status code
- The response returns a valid JSON string
- The response can be decoded to a Python dictionary
- Each element of the decoded list is a Python dictionary
- Each dictionary in the result has two keys
- Verify that the type of each key's value is correct

Remember, your services should be running and as much as possible, functional tests should be testing the end-to-end functionality of your entire app.

CHAPTER

THIRTEEN

WEEK 14: SPECIAL TOPICS

We will cover a few special topics that may help you in completing certain aspects of your final projects.

13.1 Plotting with Matplotlib

13.1.1 What is Matplotlib

It's a graphing library for Python. It has a nice collection of tools that you can use to create anything from simple graphs, to scatter plots, to 3D graphs. It is used heavily in the scientific Python community for data visualization.

Let's plot a simple sin wave

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 50)
plt.plot(x, np.sin(x))
plt.show() # we can't do this on our VM server
plt.savefig('my_sinwave.png')
```

we can keep plotting! Let's plot 2 graphs on the same axis

```
plt.plot(x, np.sin(x), np.sin(2*x))
plt.show()
plt.savefig('my_sinwavex2.png')
```

why stop now? Let's make the plot easier to read

```
plt.plot(x, np.sin(x), 'r-o', x, np.sin(2*x), 'g--'
plt.show()
plt.savefig('my_sinwavex2a.png')
```

other color combinations:

Colors:

- Blue – 'b'
- Green – 'g'
- Red – 'r'
- Cyan – 'c'
- Magenta – 'm'

- Yellow – 'y'
- Black – 'k' ('b' is taken by blue so the last letter is used)
- White – 'w'

Lines and markers:

- Lines:
  - Solid Line – '-'
  - Dashed – '–'
  - Dotted – '.'
  - Dash-dotted – '-:'
- Often Used Markers:
  - Point – '.'
  - Pixel – ','
  - Circle – 'o'
  - Square – 's'
  - Triangle – '^'

13.1.2 Subplots

using the subplot() function, we can plot two graphs at the same time within the same "canvas". Think of the subplots as "tables", each subplot is set with the number of rows, the number of columns, and the active area, the active areas are numbered left to right, then up to down.

```
plt.subplot(2, 1, 1) # (row, column, active area)
plt.plot(x, np.sin(x))
plt.subplot(2, 1, 2) # switch the active area
plt.plot(x, np.sin(2*x))
plt.show()
plt.savefig('my_sinwavex2b.png')
```

13.1.3 Scatter plots

```
y = np.sin(x)
plt.scatter(x,y)
plt.show()
plt.savefig('my_scattersin.png')
```

Let's mix things up, using random numbers and add a colormap to a scatter plot

```
x = np.random.rand(1000)
y = np.random.rand(1000)
size = np.random.rand(1000) * 50
color = np.random.rand(1000)
plt.scatter(x, y, size, color)
plt.colorbar()
plt.show()
plt.savefig('my_scatterrandom.png')
```

We brought in two new parameters, size and color, which will vary the diameter and the color of our points. Then adding the colorbar() gives us a nice color legend to the side.

13.1.4 Histograms

A histogram is one of the simplest types of graphs to plot in Matplotlib. All you need to do is pass the hist() function an array of data. The second argument specifies the amount of bins to use. Bins are intervals of values that our data will fall into. The more bins, the more bars.

```
plt.hist(x, 50)
plt.show()
plt.savefig('my_histrandom.png')
```

13.1.5 Adding Labels and Legends

```
x = np.linspace(0, 2 * np.pi, 50)
plt.plot(x, np.sin(x), 'r-x', label='Sin(x)')
plt.plot(x, np.cos(x), 'g-^', label='Cos(x)')
plt.legend() # Display the legend.
plt.xlabel('Rads') # Add a label to the x-axis.
plt.ylabel('Amplitude') # Add a label to the y-axis.
plt.title('Sin and Cos Waves') # Add a graph title.
plt.show()
plt.savefig('my_labels_legends')
```

13.1.6 Redis and plots

you can "save" your plots to Redis, however the maximum size for a key/value is 512 mb and the sum of all your data (including files) must fit into main memory on the Redis server.

```
import redis
rd = redis.StrictRedis(host='172.17.0.1', port=6379, db=0)

# read the raw file bytes into a python object
file_bytes = open('/tmp/myfile.png', 'rb').read()

# set the file bytes as a key in Redis
rd.set('key', file_bytes)
```

13.2 Storing Images in Redis

As part of the final project, your worker may create an image of a plot. If it is created inside the Kubernetes worker pod, you'll need a convenient way to retrieve that image back out of the worker container and into whatever container you curled from.

The easiest way to retrieve the image is for the worker to add the image back to the Redis db, and for the user to query the database with a flask route and retrieve the image. This would be the general workflow:

1. The user submits a curl request from, e.g., the py-debug pod to the flask api
2. The flask api creates a new job entry in the redis db, and adds the UUID to the queue

3. The worker picks up the job, and creates a plot
4. The worker saves the plot in the redis db under the job entry
5. The user curls a new route to download the image from the db

13.2.1 Initiate a Job

Imagine that when a user submits a job, an entry is created in the jobs db of the following form:

```
[isp02]$ curl localhost:5000/submit -X POST -H "Content-Type: application/json" -d '{
↪"start", "2001", "end": "2021"}'
Job 161207aa-9fe7-4caa-95b8-27f5bcbb16e7 successfully submitted
```

```
[isp02]$ curl localhost:5000/jobs
{
  "161207aa-9fe7-4caa-95b8-27f5bcbb16e7": {
    "status": "submitted",
    "start": "2001",
    "end": "2021"
  }
}
```

13.2.2 Add an Image to a Redis DB

The worker picks up the job, performs the analysis based on the 'start' and 'end' dates, and generates a plot. An example of using `matplotlib` to write and save a plot to file might look like:

```
import matplotlib.pyplot as plt

x_values_to_plot = []
y_values_to_plot = []

for key in raw_data.keys():        # raw_data.keys() is a client to the raw data␣
↪stored in redis
    if (int(start) <= key['date'] <= int(end)):
        x_values_to_plot.append(key['interesting_property_1'])
        y_values_to_plot.append(key['interesting_property_2'])

plt.scatter(x_values_to_plot, y_values_to_plot)
plt.savefig('/output_image.png')
```

Warning: The code above should be considered pseudo code and not copy/pasted directly. Depending on how your databases are set up, client names will probably be different, you may need to decode values, and you may need to cast type on values.

Now that an image has been generated, consider the following code that will open up the image and add it to the Redis db:

```
with open('/output_image.png', 'rb') as f:
    img = f.read()

```

(continues on next page)

(continued from previous page)

```
    rd.hset(jobid, 'image', img)
    rd.hset(jobid, 'status', 'finished')
```

Note: If anyone has a way to get the png file out of the Matplotlib object without saving to file, please share!

13.2.3 Retrieve the Image with a Flask Route

Now that the image has been added back to the jobs database, you can expect this type of data structure to exist:

```
{
  "161207aa-9fe7-4caa-95b8-27f5bcbb16e7": {
    "status": "finished",
    "start": "2001",
    "end": "2021",
    "image": <binary image data>
  }
}
```

It would not be a good idea to show that binary image data with the rest of the text output when querying a `/jobs` route - it would look like a bunch of random characters. Rather, write a new route to download just the image given the job ID:

```
from flask import Flask, request, send_file

@app.route('/download/<jobid>', methods=['GET'])
def download(jobid):
    path = f'/app/{jobid}.png'
    with open(path, 'wb') as f:
        f.write(rd.hget(jobid, 'image'))
    return send_file(path, mimetype='image/png', as_attachment=True)
```

Flask has a method called 'send_file' which can return a local file, in this case meaning a file that is saved inside the Flask container. So first, open a file handle to save the image file inside the Flask container, then return the image as `mimetype='image/png'`.

The setup above will print the binary code to the console, so the user should redirect the output to file like:

```
[isp02]$ curl localhost:5000/download/161207aa-9fe7-4caa-95b8-27f5bcbb16e7  > output.
↪png
[isp02]$ ls
output.png
```

Note: If anyone has a way to download the image to file automatically without redirecting to file, please share!

CHAPTER

FOURTEEN

HOMEWORK 01

Due Date: Thursday, Feb 4, by 11:00am CST

14.1 The Island of Dr. Moreau

You are Dr. Moreau and you will randomly create 20 bizarre animals. Each animal should have the following:

- A head randomly chosen from this list: snake, bull, lion, raven, bunny
- A body made up of two animals randomly chosen using the `petname` library
- A random number of arms; must be an even number and between 2-10, inclusive
- A random number of legs; must be a multiple of three and between 3-12, inclusive
- A non-random number of tails that is equal to the sum of arms and legs

Each of the 20 individual animals should be accessible from a list of dictionaries. Use the `json` library to dump your data structure into an `animals.json` file. For example, your assembled data structure may look like:

```
{
  "animals": [
    {
      "head": "snake",
      "body": "sheep-bunny",
      "arms": 2,
      "legs": 12,
      "tail": 14
    },
    {
      "head": "snake",
      "body": "parrot-bream",
      "arms": 6,
      "legs": 6,
      "tail": 12
    },
    ... etc
```

Next, create a new Python script to read in `animals.json` and print the details of one animal at random to screen.

14.2 What to Turn In

Your final homework should be turned in via GitHub. Create a repository under your GitHub account for this class. Make a subfolder called `homework01`. That folder should contain three files:

- `generate_animals.py`, which generates `animals.json`
- `animals.json`, which contains 20 bizarre animals as described above
- `read_animals.py`, which reads `animals.json` and prints one animal at random to screen

The TA will git clone your repository on the due date / time, navigate to your `homework01` folder, and inspect your code and output. The TA will try to run your code by typing `python3 generate_animals.py` followed by `python3 read_animals.py`. Additionally, `animals.json` will be entered into into a JSON validator to check if it is valid JSON.

14.3 Additional Resources

- The petname library
- Random numbers
- JSON dump
- Validate JSON with JSONLint
- Please find us in the class slack channel if you have any questions!

CHAPTER

FIFTEEN

HOMEWORK 02

Due Date: Thursday, Feb 25, by 11:00am CST

15.1 The Containers and Repositories of Dr. Moreau

This homework builds upon homework 01. You should have already written two Python scripts: one for generating a JSON file of assembled animals (`generate_animals.py`), and another for reading in and printing one animal at random (`read_animals.py`).

You are almost ready to release this code into the wild for others to use. Your aims in this homework are (1) write one new feature into the `read_animals.py` script, (2) write a unit test for your new feature in `read_animals.py`, (3) write a Dockerfile to containerize both scripts, (4) write a reasonable `README.md` for your repository.

More details on each of the requirements:

15.1.1 (1) New Feature

The `read_animals.py` script has a lot of room for improvement. The first objective is to write some new functionality into the script. Some examples of new features that could be added include:

- Pick two random animals and 'breed' them by mixing their elements to create a new animal, which is printed to screen along with its 'parents'
- Use the `argparse` library to take command line arguments and apply some error checking on the input
- Generate some summary statistics of all the animals read in and print it to screen

These suggestions are suggestions only - please be creative and write any new feature into `read_animals.py` that you want.

15.1.2 (2) Unit Test

Use the Python `unittest` library to write at least one test for your new functionality in `read_animals.py`. (Remember, a single 'test' should probably multiple 'checks' in it - see the section on Unit Testing for examples).

The test should be in a file like `test_read_animals.py` and should be executable from the command line.

15.1.3 (3) Dockerfile

The scripts we are writing should be runnable directly on ISP. They should also be runnable within a container. (Users downloading your code may want to run them directly, run them in a container, or both - it is usually good practice to provide all these options.)

Write and provide a Dockerfile that encapsulates your Python scripts. Any output generated by the scripts should be accessible outside of the container. You do not need to containerize the unit tests.

15.1.4 (4) README.md

In the `homework02` folder of your class repository, create a file called `README.md`. On GitHub, this README file will automatically render in the web interface, and it serves as the landing page of your repository (specifically of the homework02 subfolder of your repository). This is your opportunity to describe to other users what code is here, what it does, and how to use it.

Your README file should have at a minimum the following sections and instructions for users:

- A general description of the tool(s)
- Instructions on how to download and run the scripts directly
- Instructions on how to build an image with the Dockerfile provided
- Instructions on how to run the scripts inside a container
- Instructions on how to run the unit test(s)

15.2 What to Turn In

Your final homework should be turned in via GitHub. You should already have a repository for this class with a `homework01` subfolder in it. Create a new subfolder called `homework02` and put everything described above inside.

The TA will git clone your repository on the due date / time, navigate to your `homework02` folder, and inspect your code and files. We will be trying out your new feature in `read_animals.py`, we will be running the unit tests, we will be building your Dockerfile and running the scripts in the container. The instructions you provide in your README are the instructions we will follow to do those things.

15.3 Additional Resources

- Markdown Cheat Sheet
- README 101

CHAPTER

SIXTEEN

HOMEWORK 03

Filtering JSON Data by Example:

```
def get_data():
    with open('data_file.json', 'r') as json_file:
        user_data = json.load(json_file)
    return user_data

test = get_data();
print (type(test))
output = [x for x in test if x['head'] == 'snake']
```

16.1 A.

- Create 3 routes to your island animal JSON data, one has to be /animals, the other 2 should require a parameter for example:
  - /animals will return all of your animals
  - /animals/head/bunny will return all of your animals w/ bunny heads - here bunny would be a parameter
  - /animals/legs/6 will return all of your animals w/ 6 legs - here 6 would be a parameter
- Create your flask server that connects to your flask port

16.2 B.

Containerize your Flask Apps. Be sure to include your json data file in your Container

16.3 C.

Write a consumer similar to:

```
import requests

response = requests.get(url="http://localhost:5050/animals")

#look at the response code
```

(continues on next page)

(continued from previous page)

```
print(response.status_code)
print(response.json())
print(response.headers)
```

Share your url and routes to Slack, pick another student's url, and consume their data

CHAPTER

SEVENTEEN

MIDTERM PROJECT

Note: What is a UUID

first some background. A universally unique identifier is a 128-bit number used to identify information in computer systems, typically referred to as an UUID (thought, in Microsoft-speak it's called an GUID). When an UUID is generated using standard methods, for practical purposes, they are universally unique.

- UUIDS are composed of 128 bit numbers generated using standards-based algorithms that are "guaranteed" unique (i.e., the probability of collisions is so low that, to get to a 50% probability of collision, one would need to generate 2.7x10^18 UUIDs).
- There are 4 major versions of the standard - We will use UUID version 4 because: it generates uuid's with very low probability of collision without using sensitive data such as the MAC address of the server which is usually used in generating the UUID.
- The algorithms can and have been implemented in most major programming languages (yay standards!) and can generate uuid's very quickly.

How do I generate a UUID in Python?

```
>>> import uuid
>>> uuid.uuid4()

Out[1]: UUID('56849963-e90d-4322-a369-50870f0cf9fa')

# return a string:
>>> str(uuid.uuid4())
Out[2]: '66a3acf3-8009-4cd3-8d40-c8ce42229f08'
```

You are Dr. Moreau and you have an island of bizzare creatures. Now we're going to use flask to interact with our data

add the following to your JSON producer:

- a timestamp labeled "created_on"
- an unique identifier "uid"

In your Flask app,

- **have routes that**
  - query a range of dates
  - selects a particular creature by its unique identifier
  - edits a particular creature by passing the UUID, and updated "stats"
  - deletes a selection of animals by a date ranges

- returns the average number of legs per animal
- returns a total count of animals

CHAPTER

EIGHTEEN

HOMEWORK 05

Due Date: Thursday, April 8, by 11:00am CST

18.1 A.

Recall the following example pod from class:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
  containers:
    - name: hello
      image: ubuntu:18.04
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

Modify the above to create a pod in Kubernetes to print a message "Hello, $NAME", where `$NAME` is an environment variable. Give this pod a label, `greeting: personalized`.

1. Include the yaml file used and the command issued to create the pod.
2. Issue a command to get the pod using an appropriate `selector`. Copy and paste the command used and the output.
3. Check the logs of the pod. What is the output? Is that what you expected?
4. Delete the pod. What command did you use?

18.2 B.

Our first pod above was a little sad because no value was specifed for our `$NAME` variable. Let's fix that here! Let's update our pod definition to include a value for the variable, `$NAME`. We can add any number of environment variables and values using the `env` stanza inside the `containers` spec, like so:

```
---

# some stuff here...

spec:
  containers:
```

(continues on next page)

(continued from previous page)

```
    - name: # stuff...
      image: # stuff...
      env:
        - name: "VAR_1"
          value: "VALUE_1"
        - name: "VAR_2"
          value: "VALUE_2"
        . . .
```

Give the variable `$NAME` the value of your own name.

1. Include the yaml file used and the command issued to create the pod.
2. Check the logs of the pod. What is the output? Copy and paste the command used and the output.
3. Delete the pod. What command did you use?

18.3 C.

This time, let's create a deployment with the above properties, instead of a pod. We'll also add the pod's IP address to the message. Create a deployment that uses the `ubuntu:18.04` image and prints the personalized message, "Hello, $NAME from IP <pod_ip_address>", where `$NAME` is an environment variable. To add the pod's IP address to the message, we will inject another environment variable, but this time we will have k8s populate the value for us.

We can use the Kubernetes Downward API to expose pod information to itself via environment variables. Instead of using the `value` field, we use `valueFrom` with a `fieldRef` stanza that specifies a `fieldPath` property. The value of the `fieldPath` property should be a reference to the k8s property of interest.

From the k8s documentation, the following information is available through environment variables (see the docs for more details):

- `status.podIP` - the pod's IP address
- `spec.serviceAccountName` - the pod's service account name, available since v1.4.0-alpha.3
- `spec.nodeName` - the node's name, available since v1.4.0-alpha.3
- `status.hostIP` - the node's IP, available since v1.7.0-alpha.1

Thus, for example, the following code snippet included in the `env` section would create an environment variable, `$POD_IP`, with value equal to the pod's IP address (as assigned by k8s).

```
env:
  - name: POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
```

Include the following in your submission:

1. Include the yaml file used to create a deployment with 3 replica pods, and include the command issued to create the deployment.
2. First, use kubectl to get all the pods in the deployment and their IP address. Copy and paste the command used and the output.
3. Now, check the logs associated with each pod in the deployment. Does it match what you got in 2? Copy and paste the commands and the output.

CHAPTER

NINETEEN

HOMEWORK 06

Due Date: Thursday, April 15, by 11:00am CST

Complete any work needed for the in-class lab to deploy your flask API and Redis database. Include the yaml files for all of your deployments and services, and include the commands used and output generated to create each (all 5 steps).

Note: Your flask API will not work correctly on k8s after completing the steps in the lab until you modify your flask code to use the Redis service IP. There are two ways to do this, outlined below, but correctness of the flask API will not be graded as part of homework 6; the grade will based on the correctness of your k8s definitions **only**.

19.1 Updating the Flask API to use the Redis Service IP

In your flask code, you have a line that looks something like this:

```
rd=redis.StrictRedis(host='redis', port=6379, db=0)
```

Recall that the `host='<some_host>'` argument instructs the Redis client to use a particular network address (an IP address or a domain) to connect to Redis. We know from the lab that, in our k8s deployment, the Redis database will be available from the Redis service IP. We need to make sure that our flask API uses this API.

19.2 Option 1: Hard Code the Service IP Directly in the Python Code

This is the simplest approach. If our Redis service IP were `10.108.118.36` we would simply replace the above with:

```
rd=redis.StrictRedis(host='10.108.118.3', port=6379, db=0)
```

This works, but the problem is that we have to change the code every time the Redis service IP changes. It's true that we use services precisely because their IPs don't change, but as we move from our test to our prod environment (recall the discussion on environments from earlier), the Redis service IP will change. Once our code in the test environment has been tested, we want to be able to deploy it to prod exactly as is, without making any changes.

19.3 Option 2: Pass the IP as an Environment Variable

The better approach is to pass the Redis IP as an environment variable to our service. Environment variables are variables that get set in the shell and are available for programs. In python, the `os.environ` dictionary contains a key for every variable. So, we can use the following instead:

```
import os

redis_ip = os.environ.get('REDIS_IP')
if not redis_ip:
    raise Exception()
rd=redis.StrictRedis(host=redis_ip, port=6379, db=0)
```

This way, if we set an environment variable called `REDIS_IP` to our Redis service IP before starting our API, the flask code will automatically pick it up and use it.

In homework 5, you saw how to set environment variables in a k8s pod. We'll revisit this idea when discussing continuous integration.

CHAPTER

TWENTY

HOMEWORK 07

Due Date: Tuesday, April 27, by 11:00am CST

This homework assignment builds on the exercises done in class in the Messaging Systems section as well as the Week 12 material for deploying a worker to k8s. At the end of those exercises, we ended up with three files, `api.py`, `worker.py` and `jobs.py`.

20.1 A.

In the first in-class exercise from Week 12, you updated the Dockerfile for your flask application to include the new source code files in your Docker image and to include an entrypoint and a command that could be used for running both the flask web server and the worker.

1. Complete this exercise if needed and include the Dockerfile in your homework submission. Update the code to use the IP address of your test Redis service. Be sure to build the Docker image and push it to the Docker Hub.
2. With your new image on Docker Hub, create a deployment for the flask API and a separate deployment for the worker. (Creating a worker deployment was the second exercise from Week 12.) Name the files `<username>-hw7-flask-deployment.yml` and `<username>-hw7-worker-deployment.yml` and include them and the commands you used to create the deployments with your homework submission.
3. Verify that your flask API and worker are working properly: in your python debug container, create some jobs by making a POST request with `curl` to your flask API. Confirm that the jobs go to "complete" status by checking the Redis database in a Python shell. Include the following with your submission:
    a. The curl statements used and the responses (output) returned by your flask APi (these should include job id's).
    b. The Python statements (code) you issued to check the status of the jobs and the output from the statements.

20.2 B.

1. Update the worker deployment you wrote in A.2) to pass the worker's IP address in as an environment variable, `WORKER_IP`. Recall that we learned how to do this in hw 5 using the following snippet:

```
env:
  - name: WORKER_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
```

Include the updated deployment file with your homework submission.

2. In the `worker.py` file we did in class, the `execute_job` function looked like this:

```
def execute_job(jid):
    jobs.update_job_status(jid, 'in progress')
    time.sleep(15)
    jobs.update_job_status(jid, 'complete')
```

Update `jobs.py` and/or `worker.py` so that when the job status is updated to `in progress`, the worker's IP address is saved as new key in the job record saved in Redis. The key can be called `worker` and its value should be the worker's IP address as a string. Think through the best way to add this functionality in terms of the changes made to `jobs.py` and/or `worker.py`.

20.3 C.

Scale your worker deployment to 2 pods. In a python shell from within your python debug container, create 10 more jobs by making POST requests using curl to your flask API. Verify that the jobs go to "complete" status by checking the Redis database in a Python shell. Also, note which worker worked each job. Include the following with your submission:

a. The curl statements used and the responses (output) returned by your flask APi (these should include job id's).

b. The Python statement (code) you issued to check the status of the job and the output from the statement.

c. How many jobs were worked by each worker?

CHAPTER

TWENTYONE

FINAL PROJECT

Due Date: Friday, May 7th, by 11:00am CST

The final project will consist of building a REST API frontend to a time series database that allows for basic CRUD - Create, Read, Update, Delete - operations and also allows users to submit analysis jobs. **At a minimum**, the system should support an analysis job to create a plot of the data, and should be hosted on the Kubernetes cluster. Extra credit may be given if the system supports additional types of analysis jobs, deploy over multiple environments (test and prod), or has an innovative user interface.

The project will also include two separate pieces of documentation: the first will provide **instructions for deploying the system** and the second should be **geared towards users/developers** who will interact with the system.

21.1 A.

Front-end API - A set of synchronous API endpoints providing the following functionality:

- Data points retrieval endpoints
- Endpoint for creating new data points
- Jobs/graph submission and retrieval points
- Submission of other kinds of analysis jobs - extra credit

21.2 B.

Back-end workers - Backend/worker processes to work the submitted jobs:

- Worker processes framework.
- Analysis job itself (e.g., make a graph)

21.3 C.

Use of Redis database and queue structures to link front-end and back-end processes:

- Note that if the API, Workers, and Redis server might run on a different servers, you'll need to provide a configuration description (or better yet a way of propagating config)

21.4 D.

Github Repository

- Repository/code organization - The code should be organized into modules and directories that make it easy to navigate as the project grows. An example repository layout is included at:
  - https://github.com/wjallen/wjallen-coe332
  - Should contain everything needed (Dockerfile(s), Kubernetes yml files, source code, configurations, etc.) for someone else to clone the repo and deploy their own version of the app in their own namespace
- Documentation
  - Deployment docs - instructions for how an operator should deploy the system on a Kubernetes cluster
  - User docs - instructions for how to interact with your API. This should more or less be a (possibly updated version of) your HW 5/6/7, e.g.,
    - List of endpoints
    - Expected JSON responses
    - Examples of how to use your system from within curl and Python.
- Datasets

There are many public datasets available for you to to download into your project. Google is your best bet to find datasets which maybe of interest. Some examples of open datasets are https://data.austintexas.gov/ or signup for a free community account at https://data.world/ Google also has some public datasets available on the Google Cloud, https://console.cloud.google.com/marketplace/browse?filter=solution-type:dataset Some "tweaking" may be required to convert the datasets into a JSON dictionary.

CHAPTER

TWENTYTWO

ADDITIONAL RESOURCES

- Slack: https://tacc-learn.slack.com/
- Class Repo: https://coe-332-sp21.readthedocs.io/
- Canvas: https://utexas.instructure.com/courses/1299275