

Project P0

Getting ready for AggieStack

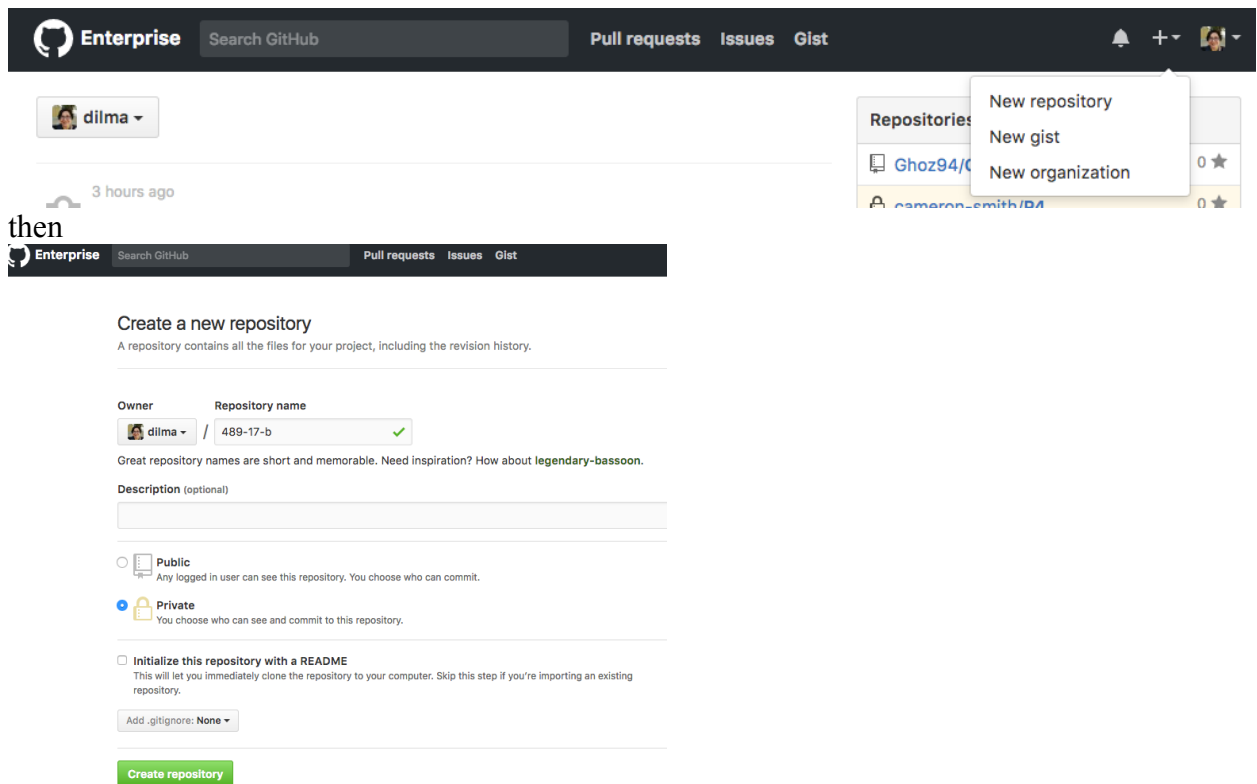
Due: 09/24/2018 11:59pm CT

By the end of the semester, you will have learned about OpenStack, a free and open-source software platform for cloud computing. In your next project – P1 - you implement some of the OpenStack functionality. P0 gives you the opportunity to start some of the work at the beginning of the semester, before you find yourself too busy.

1. Get ready to code: set up your TAMU GitHub repository

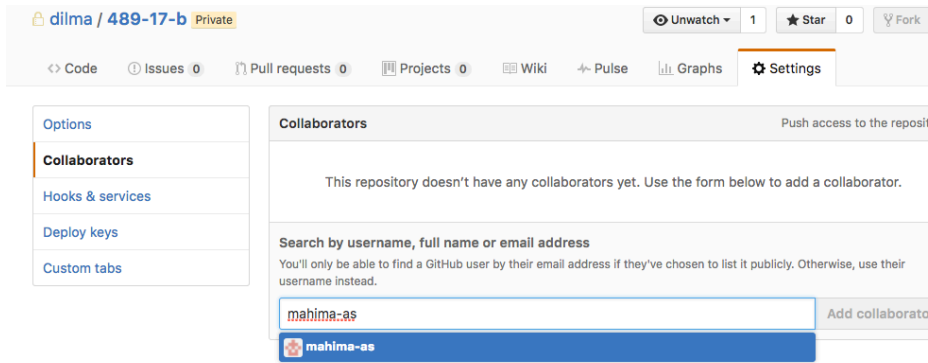
You are required to use GitHub, the version control system, while developing any of the code for this course. You need to create a repository hosted at github.tamu.edu:

- Create a private repository named 678-18-c. The figure below is from last year when we asked for "489-17-b".



The screenshot shows the GitHub 'Create a new repository' page. At the top, there's a navigation bar with 'Enterprise', a search bar, and links for 'Pull requests', 'Issues', and 'Gist'. Below this, the user 'dlima' is logged in. The main heading is 'Create a new repository' with a subtext: 'A repository contains all the files for your project, including the revision history.' The 'Owner' is 'dlima' and the 'Repository name' is '489-17-b'. There's a note: 'Great repository names are short and memorable. Need inspiration? How about legendary-bassoon.' The 'Description (optional)' field is empty. The 'Public' option is unselected, and the 'Private' option is selected with a blue dot. Below this, there's an option to 'Initialize this repository with a README' which is unselected. At the bottom, there's a dropdown for 'Add .gitignore: None' and a green 'Create repository' button.

- Add the following users as collaborators
 - “sanuj”, “bruno-cbf”
- Create a folder called “P0” for Project 0 development



Your repository needs to reflect your software development, i.e., every time you make a change to your code, you commit a version to the repository. If the modification is writing new code (for example, when you just started with an empty file), you should be committing new versions of the code often, for instance after defining a function or adding some lines of code. If your change is a code fix, you should document in the commit message what you fixed. A reasonable rule of thumb is that for short work sessions (when you write code for an hour or less), you commit when you stop the work session. For longer sessions (writing code for many hours), you may want to commit a few times, depending on how many code changes you are doing or different software pieces you work on.

2. Context for the project

We have not even discussed what cloud services are yet, but we can expect that those services (whatever they do) are available to customers through interfaces (probably through web pages, web services, and apps available for smartphones). You can assume that someone else is responsible for implementing beautiful user interfaces and for handling user authentication.

Your job is to implement the AggieStack CLI (command line interface) as specified in Sections 4 and 5.

You may feel uncomfortable working on a project that involves terms that you don't properly understand yet, such as “virtual server”, “instance”, “images” and “flavors”. This lack of familiarity is not uncommon in the initial phase of a software development project, as you are learning about your customer's domain, so aim at implementing the requested functionality even if you feel uncomfortable. By the time you go back to work on this code, later on, you will have a better understanding of the domain.

Please read this document to the end. It is a good idea to ask questions during class time.

3. Implementation environment

You can implement this project using whatever language you prefer. I would suggest using Python because it is a language used in many cloud management stacks. If you never coded in Python, you may want to use the beginning of the semester to learn and try it out. If you are using it for the first time, indicate it in your report.

Feel free to use libraries to parse arguments from the command line. If you use libraries or code ideas you find on places such as StackOverflow, make sure to indicate that in your report. Reusing software from the web is ok as long as you cite all your sources. Reusing code from colleagues or code "stolen" or "purchased" is a violation of the Aggie Honor Code (see syllabus.)

4. The AggieStack CLI – Part I

The first version of AggieStack is very primitive. The AggieStack 0.1 CLI (command line interface) implements the functionality in the table below. Your program read the command from the standard input and generates the expected output in the standard output. Error messages go to standard error. You also should append execution information to a file named aggiestack-log.txt in the current directory.

For all commands, your program should generate as output error messages for input file errors and add information to the log capturing success/failure.

Command	Description
aggiestack config --hardware <file name>	Reads the configuration file describing the hardware hosting the cloud. See Appendix 1 for details on the configuration file format.
aggiestack config --images <file name>	Reads the configuration file listing images available in the storage server. See Appendix 2 for details on the configuration file format.
aggiestack config --flavors <file name>	Reads the configuration file listing the flavor for instances available for the users. A flavor specifies a virtual machine configuration: the amount of ram, number of disks, and number of vcpus. See Appendix 3 for details on the configuration file format.
aggiestack show hardware	Output is the information about the hardware hosting the cloud in the format specified in Appendix 1.
aggiestack show images	Output the list of images available for the user to choose when creating virtual machines format???

aggiestack show flavors	Output the list of flavors available for the user to choose when creating virtual machines. format???
aggiestack show all	Output has “show” for hardware, images, flavors. You already implemented the commands you need for this.

Your implementation needs to print error messages for invalid input, e.g., non-existing configuration file, flavor, instance.

All commands generate output and update the aggiestack-log.txt log as follows:

- **Output:** none if successful, error message if there is one
- **Log:** adds command to the log and SUCCESS or FAILURE

By “updating” a log we mean that you append new information to it. If there is already a log file in the current directory, you add information to it instead of starting from an empty file.

5. The AggieStack CLI – Part II

Once your Part I API working, the next step is to add capability in your code to represent *instances* (in some platforms, the same entity may be called “guests” or “virtual machines”). Think of “instances” as virtual servers that customers can get from your cloud service. These virtual services run on top of the hardware that you have available, they use as software an “image” and how much resources they use are dictated by the instance “flavor” (or type).

Command	Description
aggiestack admin show hardware	For each physical server in your cloud infrastructure, it lists how much memory, disks, and vcpus the physical server has currently available.
aggiestack admin can_host <machine name> <flavor>	Output is “yes” if physical server <machine name> has <i>currently</i> enough memory, disks, and vcpus to host a new virtual server of type <flavor>

As with the previous commands, you need to generate the appropriate output for errors and log information on each command.

Notice that, as specified in this document, your code reads arguments provided on the command line. In order to test a lot of commands, we will have a script that invokes your program multiple times, each time to execute a single command line.

NOTE:

Please design your code to be able to handle any updates to the commands and to be able to handle additional functionalities without heavy modifications.

4. What to turn in

Submit on eCampus a zip file containing your code and a short write-up with the following information by 7/9 11:59pm Central Time:

- how much time you spent implementing this project
- show to run your code (providing a makefile to build the code if necessary)

Rubric for Project 0:

Principles guiding the grading:

- If failed to adhere to GitHub way of working: -100 points
- No instructions on how to build or run your code: -100 points

Specific submission and rubric information for Project 1 will be released with the complete Project 1 specification. Project 1 submission will not be graded without a submission for Project 0.

Appendix 1 – Hardware configuration file

The list of physical machines is provided in a configuration file in the following format:

<number of machines>
<list of machine spec one per line>

The specification of each machine has the following format:

<name> <ip> <mem> <num-disks> <num-vcpus>

where <mem> is the amount of RAM in GB, <num-disks> is the number of virtual disks the machine can accommodate locally, and <num-vcpus> is the number of vcpus (virtual cores) that the hardware can accommodate.

On eCampus, you will find a sample input file `hdwr-config.txt` illustrating the expected format.

Appendix 2 – Image configuration file

The image configuration file specifies the number of images available to be used when starting virtual machines, providing for each one the name of the image and the path in the storage server to the file containing the image. The file `image-config.txt` (available on eCampus) is an example of a configuration file in the expected format.

Appendix 3 – Flavor configuration file

A “flavor” describes a type of instance that users can request. It specifies the amount of RAM in GB, the number of disks, the number of vcpus. The file flavor-config.txt is an example of the expected format (like the other sample configuration files, it is available on eCampus).