File: manage_class.py

This is the main file that is meant to interface with the user. Through it you can create new docker containers, delete or restart an existing container, and add a new host machine to the host/port database.

parse_args():

Set up all of the possible options for argparse, including how to take their input from the command line and the info to display the main help message.

Parameters: n/a
Returns:
The parsed args of an argparse ArgumentParser object.

create_class(args, db, cli):

Create a new container for a class with the information gathered by parse_args(). This function creates a new dictionary out of the arguments, passes them to add_class.py to create a new docker container, adds the class to our database and updates the html for the landing page to include the new class.

To call:
./manage_class.py create_class -c <class_name> -r "<readable_name>" -f <first> -l <last> -e <email> -u <username> -n <hostname> -v <version> -m <memory_limit> -s <cpu_shares>
Parameters:
Args - the object returned by parse_args() that contains all of the command line input.
Db - A database object as defined by the class in database.py.
Cli - The docker-py client for the host that manage_class.py is running on.
Returns: n/a

add_host(args, db):

Adds a new host to the port/host table and makes ports 8000-8100 under that hostname available to create_class() as a valid place to host internal jupyter/rstudio servers. It then restarts nginx.

To call:

./manage_class.py -n <host_name>

Parameters:

Args - The argparse object that contains the new hostname.

Db - A database object (see: database.py).

Returns: n/a

delete_class(args, db, cli):

If the container given to argparse exists this function will delete that container, remove it from the database of classes, and updates the html to reflect the database change.

To call:

./manage_class.py -d <class_name>

Parameters:

Args - the argparse object containing the class name to delete.

Db - A database object (see: database.py).

Returns:

Prints a confirmation when the class is deleted.

create_user(user_info, cli):

Creates a new user inside of a given container with the credentials given on the command line.

To call:

./manage_class.py -u <class_name> <first> <last> <username> <email> <group>

Parameters:

User_info - The array of user information given by argparser. I.e. parse_args().user_info

Cli - The docker client for the machine this script is running on.

Returns:

Prints the status code given by the docker client after the command execution.

restart_class(name, cli):

Restart the entire docker container and turn back on the jupyterhub, rstudio and NIS clients inside of it.

To call:

./manage_class.py -r <class_name>

Parameters:

Name - Name of the class container to restart, given by argparse.

Cli - The docker client for the current host of this script.

Returns:

Logs the restart to the logs at /var/docker/log.txt.

main():

This ensures that any arguments passed in by argparse have a valid syntax and are able to be passed to their corresponding functions without error. This function also instantiates a parsed argparse object, a database object and a docker client object to be used by the functions in manage_class.py.

File: add_class.py:

This file is used by the manage_class.py file to both create new docker containers and delete existing containers based off of the information gathered by manage_class.py.

create_class(args, cli):

This is the main function of add_class.py and utilizes most of the other functions defined within it. This function uses the docker-py API to create a docker container based on the class information passed by args. It then executes a series of necessary configuration steps both inside and outside of the container.

Parameters:

Args - The argparse object given by manage_class.py.

Cli - The docker client object given by manage_class.py.

Returns:

Prints the status messages returned by each of the configuration steps.

add_html_title(cli, container_name, readable_name):

Adds the name of the container to the html page template for Jupyter. Currently not functioning and is never called.

Parameters:

Cli - The docker client object given by manage_class.py.

Container_name - The name of the current working container.

Readable_name - the human readable name for the container.

Returns:

Updated html templates within the container.

start_jupyterhub(cli, container_name):

Starts the jupyterhub client within the docker with the command line flags to disable to the need for ssl, sets the base url of the jupyter server to match what nginx is expecting and forces the uploaded configuration file to be used.

Parameters:

Cli - The docker client object given by manage_class.py.
Container_name - the name of the current working container.
Returns:
The completion status message as a string.

start_rstudio(cli, container_name):

Starts the rstudio server within the docker container by directly calling the rserver executable.

Parameters:
Cli - The docker client object given by manage_class.py
Container_name - the name of the current working container.
Returns:
The completion status message as a string.

start_ypbind(cli, container_name):

Starts the ypbind service within the container so that the NIS users can be used as users within the container.

Parameters:
Cli - The docker client object given by manage_class.py
Container_name - the name of the current working container.
Returns:
The completion status message as a string.

add_base_url(cli, container_name):

Adds the base url for jupyterhub to its configuration file inside of the container so that it serves information to nginx correctly. Depreciated by the fact that this function can be called using flags on jupyterhub startup.

add_home_directory(cli, container_name):

Adds a line to the configuration file of jupyterhub within the container to set the location of the home directory to the corresponding class file on /ACTF.

Parameters:

Cli - The docker client object given by manage_class.py
Container_name - the name of the current working container.
Returns:
The completion status message as a string.

add_group_whitelist(cli, container_name):

Adds users that belong to either the group with the name of the container or the cgrb group to jupyterhub's authentication whitelist.

Parameters:
Cli - The docker client object given by manage_class.py
Container_name - the name of the current working container.
Returns:
The completion status message as a string.

create_instructor(cli, args):

Creates a user within the container by calling the manage_users.py script and passing it the information from arg parser. This function was deprecated by the addition of NIS users.

write_nginx_config(class_name, jupyter, r_studio):

Creates a new nginx configuration file outside of the container and adds the necessary configuration as dictated by the jupyter and r_studio strings.

Parameters:
Class_name - The name of the current working container/ class.
Jupyter - The multi-line configuration string as returned by get_nginx_jupyter_config().
R_studio - the multi-line configuration string as returned by get_nginx_r_config().
Returns:
A valid nginx configuration that incorporates the newly created class.

get_nginx_jupyter_config(host, port, class_name):

Takes the networking information about the class and returns a string of valid nginx configuration settings for jupyterhub.

Parameters:

Host - The hostname of the machine that will be running the docker container.

Port - The port on that host that jupyter will be communicating with nginx through.

Class_name - The name of the current container/class being created.

Returns:

A multi-line string of nginx configuration.

get_nginx_r_config(host, port, class_name):

Takes the networking information about the class and returns a string of valid nginx configuration settings for rstudio.

Parameters:

Host - The hostname of the machine that will be running the docker container.

Port - The port on that host that jupyter will be communicating with nginx through.

Class_name - The name of the current container/class being created.

Returns:

A multi-line string of nginx configuration.

delete_class(name, cli):

Deletes the class container that matches the given name and removes all a associated configuration files from nginx as well.

Parameters:

Name - The container/class name of the class to be deleted.

Cli - The docker client object as given by manage_class.py.

Returns:

n/a

valid_input(input_string):

Determines if the given input_string is of a valid format to be accepted as input for a new class or user. A valid format means it only contains letters, numbers, @s, periods, underscores, spaces or apostrophes.

Parameters:

Input_string - The string that is to be analyzed for validity.

Returns:

A boolean of the input_strings validity.

File: class_database.py

This is the class definition for the database object that is used to manage and keep track of the containers created by these scripts. This database uses sqlite3 and contains two tables. The first table holds all of the information that was given at the time of a containers creation. The second table holds the port information about the machines that are hosting these containers and keeps track of which host has available ports when a new container is to be created.

__init__(self):

Sets the path to the database as /data/cgrb/database.sqlite and defines the names of some of the columns. The constructor then creates both a port table and a class table if they have not yet been created.

get_connection(self):

Attempts to connect to the database at the given path in order to open a line for its mutation.

Returns:

A valid sqlite3 cursor.

create_class_table(self):

If the table does not exist it creates a new table with the name classes. This table has columns host, port, class_name, readable_name, instructor, first, last, email. The host and port columns are foreign keys from the the port table.

Returns:

An updated database that is guaranteed to have a table of this format.

populate_port_table(self, hv):

Adds a 100 new lines to the port table for the ports 8001-8101 under the given hostname.

Parameters:

Hv - The new hostname.

Returns:

100 new available ports in the port table.

Create_port_table(self):

If the port table does not exist then create it. The port table will have the columns host, port and in use. Host and port are both defined as primary keys and therefore there can never be two of the same port on the same host in this table.

Returns:

A database that is guaranteed to have a port table.

insert_class(self, host, port, class_name, readable_name, instructor, first, last, email):

Inserts a new line into the class database with the given information or updates the line if there is already an entry with the given class name. The port table is then also updated to reflect the given combination of host and port to be in use.

Parameters:

Self-explanatory class information as given by argparse in manage_class.py.

Returns:

An updated database that now includes this class's new or updated information.

get_instructor_info(self, class_name):

Get's the information about the instructor of a class given the class name. This information is then used to update the html for the main landing page when a new class is created.

Parameters:

Class_name - the name of the class that has the instructor's information.

Returns:

A dictionary containing the instructors information under the keys 'first', 'last', 'email' and 'class_name'.

get_class_names(self):

Returns a list of all the class names that are currently in the classes table.

get_unique_hosts(self):

Returns a list of all the unique hostnames currently in the port table.

remove_class(self, class_name):

Frees all of the ports that are currently in use by the given class name and then deletes the row under that name from the class table.

Parameters:

Class_name - name of the class to delete.

Returns:

An updated database excluding that class with its ports freed.

print_ports(self):

Print all of the ports in the port table to stdout.

print_classes(self):

Print all of the classes in the class table and their associated information to stdout.

get_available_ports(self, host):

Returns a list of all of the available ports on the given host.

Parameters:

Host - The hostname to find an available port on.

Returns:

List of available ports on host.

File: html_manager.py

write_html_class(class_name, readable_name, instructor_first, instructor_last, instructor_email):

Creates a string of html that contains the class information that is passed to this function. This html string is then written to the index.html of the main landing page to append the current list of classes there

Parameters:

Class_name - name of the class.
Readable_name - the main name to put as the title of the class.
Instrucor_first - first name of instructor.
Instructor_last - last name of instructor.
Instructor_email - email of instructor.

Returns:

An updated nginx index.html to reflect the change to the class list of containers.

write_html_head():

A function that copies all of the html before the list of classes back into the index.html when it is updated by this module.

write_html_tail():

Contains a hardcoded string of html that is meant to create the final logo at the bottom of the page and wrap up the html tags.

update_html():

Updates nginx’s index.html so that the class list on the webpage accurately reflects what the actual class list in the database is. This is called everytime a container is created or destroyed by manage_class.py.