Project 3 - Section 01
Group 10: Bryan Mendoza-Trejo, Eduardo Gomez, Cristian Rodriguez

Project Description
Project 3 includes the following three services: likes.py, polls.py, serviceRegister.py. The likes.py enables users to like posts, view users likes, and see top likes. The polls.py enables users to create polls, find polls by polls_id, view vote results, and vote in polls. The likes.py uses Redis, a key-value store, to contain all the likes of posts that users have liked. Polls.py uses Amazon's NoSQL key-value store service DynamoDB to store all the polls each user creates. The service registry manages the service instances of each web service. The hug startup decorator is used within each service that registers each service instance into the service registry database. Health checks are managed in the service registry. A health check verifies that each service instance is operational with a status code of 200 OK. Once the service registry API is utilized, the health checks occur periodically (every thirty seconds), which check all service instances are still accessible. The periodic checks are handled with python's threading libraries in which locking resources occur when checking each service.

How to create the databases and start the services
1. Run **./bin/init.sh** on the project terminal to create the databases.
2. Close the dynamoDB connection upon success **(Ctrl+C)**.
3. **Foreman start** to run all microservices.

<u>Note</u>: To include the haproxy file you must follow these steps:
On the project directory terminal run:
1. sudo cp -f haproxy.cfg /etc/haproxy/haproxy.cfg
2. sudo systemctl restart haproxy

Command to view the file: sudo nano /etc/haproxy/haproxy.cfg

polls.py
The polls.py program utilizes the hug API framework to create custom REST API directives. The database for the polls service was created with Amazon's DynamoDB NoSQL service. Polls utilizes common CRUD functions to access the key-value store data found

in the local DynamoDB database. Note: Our HAProxy server is configured to host the likes REST API on port 5400 (localhost).

on_start_up()

Registers the service URL including the path to the service instance in the service registry on startup.

health_check()

Checks which service instances within the web service are available and healthy to the client; 200 OK.
Example:

http GET localhost:5400/health-check

get_results(polls_id, choice)

Get the results from a users poll. The API asks for the polls_id of the poll and the choice to see the result of that choice. To get the results of the poll, the get_results_id function was used which queries the polls_id and response to get the particular item by the polls_id key and add the item to the Polls table.
Example:

http GET localhost:5400/vote/results?polls_id=”462d7f16-ffb4-4762-8333-1270b76995a2” choice=”yes”

votePoll(user, polls_id, choice)

Allows the user to vote in the poll. The API asks for the username, polls_id of the poll, and the vote choice. Our API queries the polls, voters, and results tables. The API checks if the user has already voted on this particular poll, if the poll exists, and if the response for this poll exists. Once the correct poll and response has been located, we use the increase_votes function to increase the total response and total votes by one. If increasing the votes is successful, the API puts the poll results in the table along with inserting the choice into the votes table.
Example:

http POST localhost:5400/poll/vote?user=”chris” polls_id=”462d7f16-ffb4-4762-8333-1270b76995a2” choice=”yes”

get_polls(polls_id)

The get_polls API retrieves the poll given a polls_id. If the poll does not exist, it will return an error.

Example:

http POST localhost:5400/polls?polls_id=”462d7f16-ffb4-4762-8333-1270b76995a2”

createPoll(polls_title, username, responses)

Creates a new poll. The API asks for the polls_title for the poll, username of the user, and responses as a comma delimited list. The put_polls function is used to insert the poll into the Polls table on DynamoDB.

Example:

http POST localhost:5400/create/poll?polls_title=”Italian food or Mexican Food?” username=”chris” responses=”Italian”,”Mexican”

likes.py

The likes.py program utilizes the hug API framework to create custom REST API directives. We also use the sqlite-utils library to retrieve user posts. We also use the redis library to store total likes, posts that a user liked, and the top posts with the most likes. Note: Our HAProxy server is configured to host the likes REST API on port 5100 (localhost).

on_start_up()

Registers the service URL including the path to the service instance in the service registry on startup.

health_check()

Checks which service instances within the web service are available and healthy to the client; 200 OK.

Example:

http GET localhost:5100/health-check

like(username, posts_id)

Creates a new like for a post. The API asks for the username of the user, and posts_id of the post they want to like. The like function also stores all the information such as the number of likes, the top posts with the most likes, and the post a user likes on redis in the background.

Example:

http POST localhost:5100/new/likes?username="John"&posts_id="1"

likes(posts_id)

Get the number of likes for a post. The API asks for the posts_id. The API returns the number of likes for a post and the posts_id.

Example:

userLikes(username)

Get the posts id a user likes. The API asks for the username. The API returns all the posts ids the user has liked and the username of the user.

Example:

http GET localhost:5100/user/likes?username="John"

topLikes()

The API returns the top ten posts with one being the most likes and ten being the least likes.

Example:

http GET localhost:5100/top/likes

serviceRegister.py

This service consists of health checks, multithreading, locking and a dictionary. The health checks are used to check the status of our connections such as 200 status codes. Since multithreading is required, we need locking to avoid race conditions. Finally a dictionary is used to store the url of the services and their status codes.

on_start_up()

Initiates the thread function.

services()

Get All services information including service_name, url, http, and id from the services database.

Example:

```
http GET localhost:5500/services
```

newService(service_name, url, http)

Creates a new service. The API asks for the service_name, url, and http.Utilized mainly in the on_start_up functions to create the new service.

Example:

http POST localhost:5500/new/service?service_name=”serviceRegistry”&url=”http:localhost:5500”&http=”GET”

serviceExists(service_name, url, http)

Verifies the service information including service_name, url and http from the services database. Utilized mainly in the on_start_up functions to check if the service exists.

Example:

http POST localhost:5500/service/exists?service_name=”serviceRegistry”&url=”http:localhost:5500”&http=”GET”

healths()

Utilized mainly in the health_check functions for each service to get all health information for that service including servic_name, url, http, status_code, and id from the healths database.

Example:

http GET localhost:5500/health-check?serive_name=”polls”

urls_health(service_name, http, url, status_code)

Creates a new health of a url. The API asks for the service_name, http, url, and status_code. Utilized mainly in the thread function to create the new health of a url.

Example:

http POST localhost:5500/urls/healths?service_name=”serviceRegis *try”&http=”GET”&url=”http:localhost:5500”&status_code=”200”*

health_check(url)

Verifies the health of the url from the healths database and verifies if that url exists in the service registry from the services database. If the status_code of the url is 200 then it does nothing if it's not 200 OK then it deletes it from the services and healths database. Utilized mainly for thread lock function.

thread_lock(object)

Locks one service url at a time to do the health_check function for that url and then unlocks once finished.

thread_fucntion(lck)

Stores the service_name, url, and http into three separate lists to get the status_code for each url and store it into the healths database. At the same time performs the lock function for one url at a time. Once finished it sleeps for 30 seconds and starts the thread again to continuously check the health.