sgcli

A sweetgreen CLI for ordering from Sweetgreen

Created by: Zack Elliott, Alessandro Portela, Raghav Joshi

Github: https://github.com/zelliott/cis191-project (https://github.com/zelliott/cis191-project)

To Run:

Note: Most of these commands will kick off a series of prompts for you to enter information.

Add an account:

```
$ ./sgcli.sh --addAccount
```

Change any account information:

```
$ ./sgcli.sh --changeAccount
```

Remove your account information:

```
$ ./sgcli.sh --removeAccount
```

Place an order:

```
$ ./sgcli.sh --order
```

Place a scheduled order:

```
$ ./sgcli.sh --scheduledOrder
```

Remove a scheduled order:

```
$ ./sgcli.sh --removeScheduledOrder
```

Project Description / Goals:

Our team has built a command-line program for ordering from Sweetgreen. This program includes the following major components:

- **Restaurant Location:** Logic that either accepts a particular preferred location to order from, or helps the user find the nearest Sweetgreen.

- **Order Processing:** Our program must process the order and interact with Sweetgreen's online checkout portal.
- **Secure Payment and Authentication:** Our program will need to securely store all payment and authentication information on the user's computer.
- **Scheduled Orders:** The ability to create a cron job to submit an order at a particular date/time.
- **Command-Line Interface:** An easy to understand CLI for specifying all aspects of ordering. For example, users will be able to specify menu items (e.g. salad types, sides, drinks, etc...), specific customizations, as well as other details of their order (e.g. type of payment, location, special instructions).

Notes:

Our project proposal changed to ordering from Sweetgreen from ordering to Chipotle because Chipotle's online ordering portal was offline for maintenance this last weekend. Given that our group was not sure when the portal would be back up, we decided that it would be a smart idea to switch to a different restaurant. We chose Sweetgreen because, although they do not have an "official" API, many of the requests being made on their website are prefixed with `/api/`, and thus it seemed like we would be able to identify which to use.

Additionally, our team has run into some unexpected difficulties placing the final order with Sweetgreen. Sweetgreen expects that the payment information and the user credentials are encrypted in a certain way. We are working on reverse-engineering exactly how to create our request payload so that Sweetgreen validates and accepts it.

Project Structure:

Overall program flow can be visualized as follows. Each individual component is described in more detail below:

1. User calls program: `$ ./sgcli.sh [some command flag]`
2. Corresponding action script is run: `sgcli.sh --> actions/[some action]`
3. Action script sends data to a Python API endpoint: `actions/[some action] --> api/[some request for data]`
4. API endpoint pipes data to central Python process: `api/[some request for data] --> sgrunner.py`
5. Central Python process makes a request to Sweetgreen's API: `sgrunner.py --> sgorder.py`
6. Order class returns data from Sweetgreen to central runner: `sgorder.py --> sgrunner.py`
7. This process pipes response back to API endpoint: `sgrunner.py --> api/[some request for data]`
8. Action script grabs output from API endpoint: `api/[some request for data] --> actions/[some action]`
9. User is prompted for additional input, and steps 3-8 continue until action is completed.

sgcli.sh

Our program runs through the shell script `sgcli.sh`, which kicks off all other helping programs/scripts. First, this script sets up two ***named pipes*** called `/tmp/sgcli-send` and `/tmp/sgcli-receive`. These pipes are used by our various Python scripts to communicate to one another. Second, it kicks off `sgrunner.py` in the background. Third, it handles all user I/O in our program. This involves (1) parsing command line arguments such as `--order` and `--addAccount`, (2) reading user input and sending it to the various Python scripts, and (3) validating user input and displaying success/error messages. The various actions/commands that the user can specify have been factored out into the `actions/` directory, for better separation. Fourth, it handles creating and removing cronjobs for scheduled orders. Finally, this script cleans up all processes and pipes it creates at the end of running.

sgrunner.py

This Python script is called by `sgcli.sh`, and essentially just loops in a non-blocking read loop, reading input from the pipe `/tmp/sgcli-send`. Whenever input comes in, it parses the input to determine what it should do, and what it should send back to the pipe `/tmp/sgcli-receive`. For example, if the input `--getLocations 19104` is sent to `sgrunner.py`, then the program parses `--getLocations` to determine that it should return a list of restaurant locations specific to that particular zip code. Thus, this script can be thought of as the middleman between `sgcli.sh` (which handles all I/O with the user) and `sgorder.py` (which actually performs all of the hard work of making requests to Sweetgreen's API).

sgorder.py

As mentioned above, this script holds all logic regarding making requests to Sweetgreen's various endpoints, parsing response JSON, and sending that data back to `sgrunner.py` to eventually be piped back to `sgcli.sh`.

api/

These Python programs are simply helper methods used by `sgcli.sh` and `sgrunner.py` to send data back and forth through the pipes. They're not super interesting.

actions/

These shell scripts hold all of the logic for the various user I/O processes that are managed by `sgcli.sh`. Depending on what flags our program is called with (i.e. `--order` vs `--addAccount`), a corresponding script from this `actions/` directory is run.

saver/secureSaver.py

This Python program handles all aspects of saving and encrypting sensitive account data to the user's computer. When it is called, it first creates a folder & file `.sgcli/data.json`. This JSON file will hold all account information needed by our program to submit an order to Sweetgreen (i.e. Sweetgreen account information, credit card data, etc...).

The class works in the following manner. First, the JSON data blob is stored in the file along with a required password. The password is hashed using SHA512 with a random salt, and both are included in the JSON blob. Then, whenever information needs to be retrieved from the `data.json` file, a password is required, and the SecureSaver class verifies that the hashed passwords match. If so, it returns the data. Whenever information needs to be stored in the file, again, a password is required, and the class ensures the hashes match. If so, it updates the blob accordingly.

Additionally, all data other than the password and the salt are encrypted using a basic XOR encryption (not super secure, but more of a proof of concept). This means that if a user opens the `data.json` file, then they will be unable to read any sensitive data (e.g. credit card numbers will be jumbled).

Zack:

I worked on a number of different pieces, listed below:

- I helped to plan out the general strategy / program flow for this project. There are a number of different moving pieces (multiple scripts running, making requests to an external API, creating/destroying files, etc...) that needed to be synchronized together properly.
- I helped to create the central `sgcli.sh` runner, including adding logic to read user input, create pipes and spawn background processes, create/destroy cronjobs, validate user input, etc...
- I created the SecureSaver class in `saver/` to handle securely storing sensitive user information on one's computer. This turned out to be more difficult than I initially envisioned.
- I created the logic to create and remove cronjobs.
- I created the helper functions in `api/` to for our shell script to communicate with the Python class actually interacting with Sweetgreen's API.

Alessandro:

I focused on the central shell script-- specifically, on implementing user I/O: parsing script flags and implementing the various flag cases, parsing and storing command-line user input, and validating user input and displaying success/error messages.

I also implemented the get_times Sweetgreen API call in sgorder.py, along with its corresponding parsing and formatting in its helper function in sgrunner.py. Get_times gets a store's hours, given a restaurant-id, and outputs a list of possible order pickup times based on those store hours.

Raghav:

I wrote the Order class contained in sgorder.py (originally named scraper.py) which performs all of the Sweetgreen API calls:

- Given a zip-code, gets the closest sweetgreen locations nearby (provides JSON object containing IDs, addresses, and restaurants)
- Given a restaurant-id, gets the menu for that particular restaurant
- Given the user's information (user, password), logs into the account and provides a

valid CSRF-token to maintain state across the session

- Given what was selected by the user (salad, soup, etc), building the line-item / order and creating an order token for checkout