Objective:

Sometimes projects need to be refactored, both in terms of technical architecture and human process. The LOST project is going to undergo this kind of change.

I'm really thankful for the feedback I've been getting regarding the level of difficulty and level of uncertainty regarding what is being asked for in the assignments. I'm sorry it has taken so long to get the message regarding the process problems and make a shift in strategy. Going forward, the assignment will have a more classical assignment structure and steps will properly align with how scoring will be done.

The first assignment of our remaining 5 iterations is to archive the work done and establish a new baseline to build from. The new baseline will be a simple web application that provides a way to add users, login, logout, and an empty dashboard to show that login worked.

At the end of each step instructions are the points associated with the step. **Grading will be done in stepwise order and a completely failed step will cause grading to stop.** Side effects of this are that you must have the correct branches and file system structure; failing before the low water mark is passed results in 0 points. There is a piazza thread titled "HW6 Startup Script" where Andy gives details on how your code will be installed and provides the script he will be using to install your code. It would be wise to test your work using this script since we will no longer debug deployment and syntax issues during grading.

Step 1 - Archive the existing work in a branch

A lot of work has been done already. This should be kept somewhere so that you can look back at the code when you encounter similar issues in the refactor/redesign.

You will need to create a new branch named *'lost_archived'* containing the current work and push it to the server.

- Create the new branch:

  ```
  git branch lost_archived
  ```

- Change to the branch:

  ```
  git checkout lost_archived
  ```

- Push the branch:

```
git push origin lost_archived
```

-- low water mark - 0 pts

Step 2 - Setup project directory structure in the master branch

The new development directory structure will look very much like the old development structure. You will have a preflight.sh script to setup your database tables and copy your code to the correct places. Database code/scripts will be kept in a directory named *'sql'*. Source code will be kept in a directory named *'src'*. You may want a *'util'* directory to hold utility scripts and a *'clients'* directory to hold web service clients. Each directory should have a README.txt that enumerates what the directory contains and briefly describes its purpose.

- Change back to the master branch:

  ```
  git checkout master
  ```

- Remove existing files from the filesystem and version control:

  ```
  git rm
  ```

- View file/directory changes are being tracked:

  ```
  git status
  ```

If you choose to get rid of all non-essential existing work your repo should look like:

```
$REPO/
    README.txt
    preflight.sh
    sql/
        README.txt
    src/
        README.txt
```

Commit the updated master branch contents to github. Use the GitHub web interface to verify that the changes have been committed.

- Create a new branch for assignment6:

  ```
  git branch assignment6
  ```

- Change to the branch:

  ```
  git checkout assignment6
  ```

- Push the branch to GitHub:

```
git push origin assignment6
```

All of the assignment6 work will be done in the assignment6 branch and merged into the master branch at the end.

-- Minimal deliverable - 5 pts (all or nothing)

Step 3 - Setup the new database

The new application will handle user authentication internally. To do this, at least one table will be needed to hold username and password pairs. You may look/think ahead and add additional tables. Usernames will be no longer than 16 characters. Plaintext passwords will be no longer than 16 characters. If you plan to store hash passwords (optional), you might want to look up the length of the hash output so that you can set the column width.

Create a file named *'create_tables.sql'* in your SQL directory that has the needed create table statement(s). Have your preflight script call your create_tables.sql to setup the database tables. Test your preflight script.

Add comments near the create table statements to briefly explain your design choices. Did you choose to use the username as the primary key directly or did you choose to add a numeric primary key? Why is the password field the length you chose? etc?

Commit and push your code.

-- Database dependence - 6 pts (2 for create table, 4 for comments)

Step 4 - Create user screen

The new application will require a few screens. The first of these is the create user screen, which will be used to create users for testing.

Your app.py should get the database connection information from a file named lost_config.json located in '/home/osnapdev/wsgi'. The lost_config.json file has the same format as the configuration file used in earlier weeks. You may use the instructors config.py code to read the lost_config.json file. If the configuration file does not work, we will not be able to grade this step or any later steps.

1. Create the app.py file in the src directory.

2. Add a route for '/create_user'
3. If the request is a 'GET' type request:
    - A template with a form that allows a username and password to be submitted should be rendered.
    - The form method should be POST and action should go to the /create_user route.
4. If the request is a 'POST' type request:
    - The username and password should be read from the form
    - Check if the username already exists
        - If the username exists, render a page to let the user know the username already exists
        - If the username does not exist, insert the username and password data and render a page to let the user know the user was added successfully

If you use mod_wsgi, update the preflight script so that app.py will install correctly. Test your create user screen using your web browser and psql to check for the newly created user. Commit and push your code.

-- Minimum application code -- 3 pts (all or nothing)

Step 5 - Create login screen

1. Add a route for '/' and '/login'
2. If the request is a 'GET' type request:
    - A template with a form that allows a username and password to be submitted should be rendered.
    - The from method should be POST and action should go to the /login route.
3. If the request is a 'POST' type request:
    - The username and password should be checked against the database.
    - If the username and password are matched, redirect the user to the /dashboard route
    - Otherwise render a page letting the user know the username and password are unmatched

Test your login screen using your web browser. You will get a 404 error for the dashboard since it hasn't been written yet. Commit and push your code.

-- Minimum functionality -- 3 pts (all or nothing)

Step 6 - Create dashboard screen

Add a route for '/dashboard'. The page should only render for 'GET' type requests. The dashboard should display the username of the currently logged in user.

Test your login screen using your web browser with a valid username and password. Verify that the username is displayed on the dashboard. Verify that an invalid username and password does not go to the dashboard. Commit and push your code.

Merge the assignment6 branch into the master branch:

```
git checkout master
git merge assignment6
git push
```

-- Complete functionality -- 3 pts (all or nothing)

Use the GitHub web interface to verify that the expected files are present.

FAQ

- **preflight.sh only calls create_tables.sql?**
  The depends on wether you use mod_wsgi to the application or not. Next assignment you will be required to use mod_wsgi, so I would suggest trying to get that working if you have time. If your app.py is run directly, preflight.sh only calling create_tables.sql is sufficient. If you are using mod_wsgi, your preflight.sh script should also copy the needed files to $HOME/wsgi.
- **Will preflight.sh stil be taking in a database name?**
  Yes. The preflight.sh script will still recieved the database name as the first argument.
- **What should be on the '/' page?**
  The '/' page and '/login' should produce the same result. This can be done by repeating code, redirecting to '/login', or associating two routes with the function to render the login page.
- **Can/should we use sessions?**
  I believe sessions will be required to get the desired functionality. The username from the login must somehow reach the dashboard.
- **Should passwords be unique?**
  No. Usernames should be unique. Multiple users could use the same password but hopefully they wont.