CS6460 Project Catalog

The Chrome plugin is available as an unlisted extension here. The code for the project is outlined in the following sections. Each heading corresponding to the respective directory relative to the package root.

api

This directory contains the actual backend service. This is a Flask application written in Python 3.6. It uses PeeWee ORM for managing the database and relational models. TextBlob is used for NLP. Everything else is hand-rolled for speed and simplicity.

The layout is fairly straightforward. The entry point is standard for Flask apps and is location in app.py. This file sets up the database connection and defines all of our routes. Had this been a larger backend, the routes would have been segregated but I kept the routine simple by only exposing three REST endpoints.

Resources

The resources package contains all of the data models that are managed by the ORM.

- Courses: Maps course name to Piazza id. The ID is what you see in the URL, also called the network id. The id for CS6460 is j6azklk4gaf4v9.
- Stats: Holds all the statistics for a student's online engagement. As you can see, there is nothing directly identifying students and most of the content is simply calculated or aggregate in nature.
- Scrape: This module performs the important task of actually scraping the data from Piazza and processing into into a Stats model. The PiazzaPost model is an intermediate form that is used for passing structure data between the parser and the NLP processor.
- User: Maps a user to their authentication secret and the courses that they are enrolled in. This too is light on identifiable data and only stores the Piazza student ID and a random secret that has no derivation from the user identity.

Common

The common lib holds all the modules that are not specific to web requests. This includes the low-level database, datetime handlers, and of course the NLP. The NLP

modules does most of the interesting calculations. The CSS gradient stuff is all handled in the util module.

The database is interesting because this had the highest cost potential. The naive storage method saw hundreds of MBs of storage without any clear purpose. The refined version brought this down to a few hundred KBs with much less post-processing effort. A double-win is that we are no longer storing Piazza information, only composite data. With this small amount of data, we solve the problem of updating user data because we can simply dump everything and
recalculate as often as needed. We're only storing numbers that are meaningful to us so even if someone were to get the database, the data is worthless.

Overall, I'm very proud of this module. It is well tested and well documented. There will of course be things I'd like to refactor but those are more stylistic changes that fundamental transgressions.

api_test

This is the unit test suite in which we achieved 91% code coverage. Emphasis was placed on accuracy of the date methods and Piazza content extraction as these are critical components. The actual Piazza scraping is not tested as that is a 3rd party library. I've mocked this service out were appropriate to facilitate regression testing. Also, because I'm nice person I've built in a random throttle so this service does not knock Piazza over during our scrape sessions.

Unit testing uses the Python unittest package and there isn't much more to it.

chrome

The chrome plugin lives in this directory. As described in the manifest, it requires access to your tabs, cookies, and storage. I use the tabs API to determine the course name and ID. I use the cookies API to locate the Piazza cookie that contains the user id string. I then use the storage API to persist the user authentication tuple. This is nice because it follows your browser without any extra effort.

For the UI, this is using Materialize CSS and some of my hand-rolled classes. For javascript, I keep things simple and avoided the nightmare that all these modern frameworks have become. I did indulge a bit with jQuery though because that makes the code much more concise. One exception is my use of chart.js because

there is not point in making your graph library. Chart.js is simple and free so I use that for generating the pie charts.

The only Chrome specific code is located in popup.js and background.js. The latter is used as a listener to enable and disable the extension based on if the user is on a Piazza website. Popup.js does all the authentication, ajax, and rendering. Due to the nature of Chrome extensions, it is infinitely easier to keep as much of your logic in a single file as possible. Had this been a normal project, the code would certainly be more spread out.

Popup.html is barebones as I generated most of the html dynamically. What is does provide however, are the base div anchors and library imports so the Chrome extension doesn't holler about security. This is actually why I host my own libs instead of using CDNs. Google is very strict about cross-site scripting and for good reason.

To load the development version of the plugin, refer to the Google developers page: [https://developer.chrome.com/extensions/getstarted#unpacked](*https://developer.chrome.com/extensions/getstarted#unpacked*)

private

Move along, nothing to see here.

Tools

This is my hacking ground for mocking, testing, experimenting, and making things bend to my will. The main module provides a command line interface for a variety of tools that I used to explore Piazza, parsing, and the most efficient ways to store my data. THe project was bootstrapped from the early work done in this module.

Ready to Build?

See setup_uwsgi.md to get started with the server.