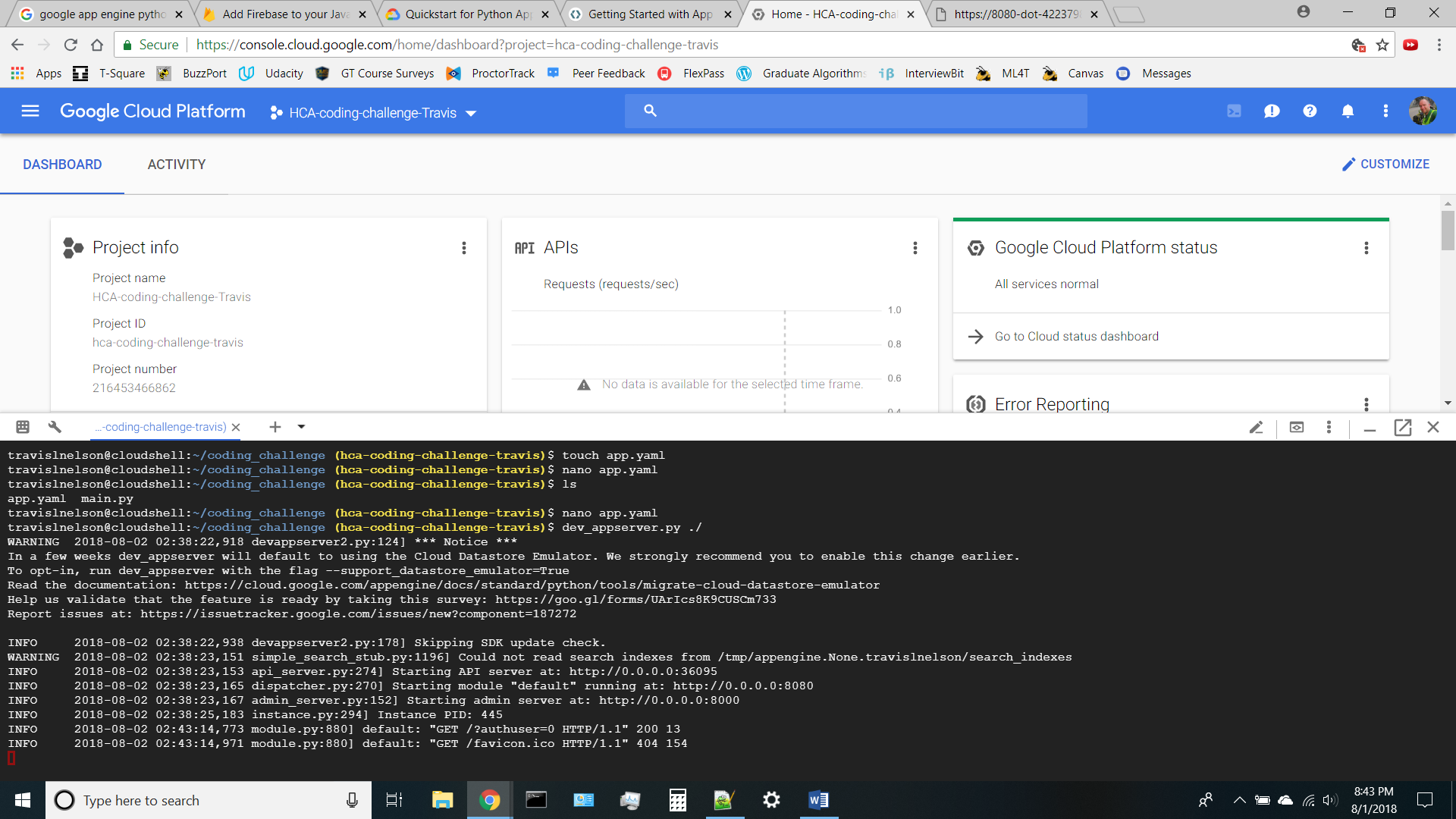
This document describes how to create a simple Web App for the HCA coding challenge. I chose to make a Web App using Python in Google App Engine, which provides both the frontend and backend of the application all in one convenient package. The backend is a Google Cloud Datastore, which is a NoSQL database.

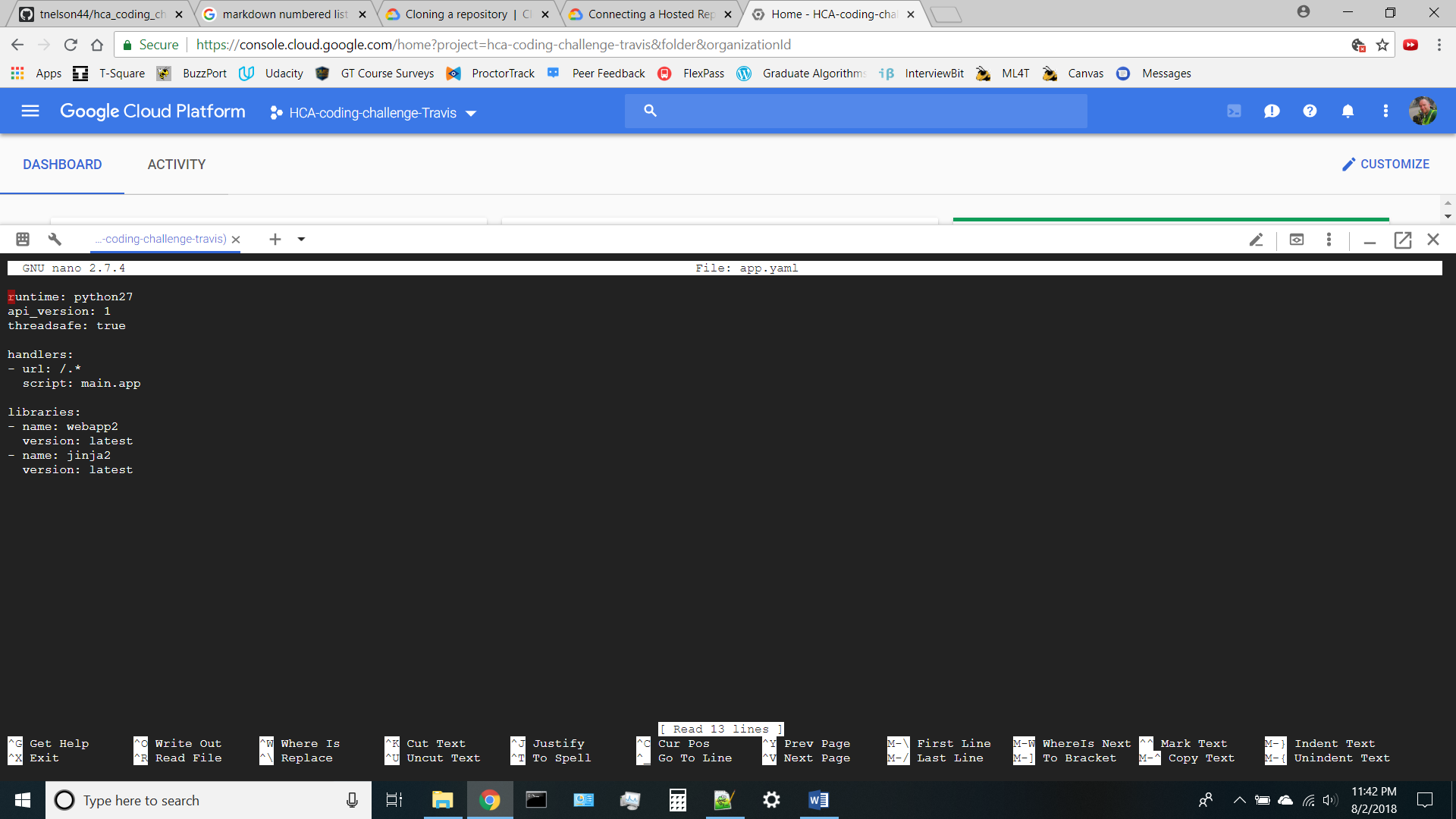
The url for this Web App is: https://hca-coding-challenge-travis.appspot.com

**Setup:**

1. Create a Google account if you don’t already have one: <https://accounts.google.com/SignUp>
2. Once you have a Google account, go to their cloud platform: [console.cloud.google.com](http://console.cloud.google.com/) and create a new project
3. Start a shell by clicking on the button in the toolbar:

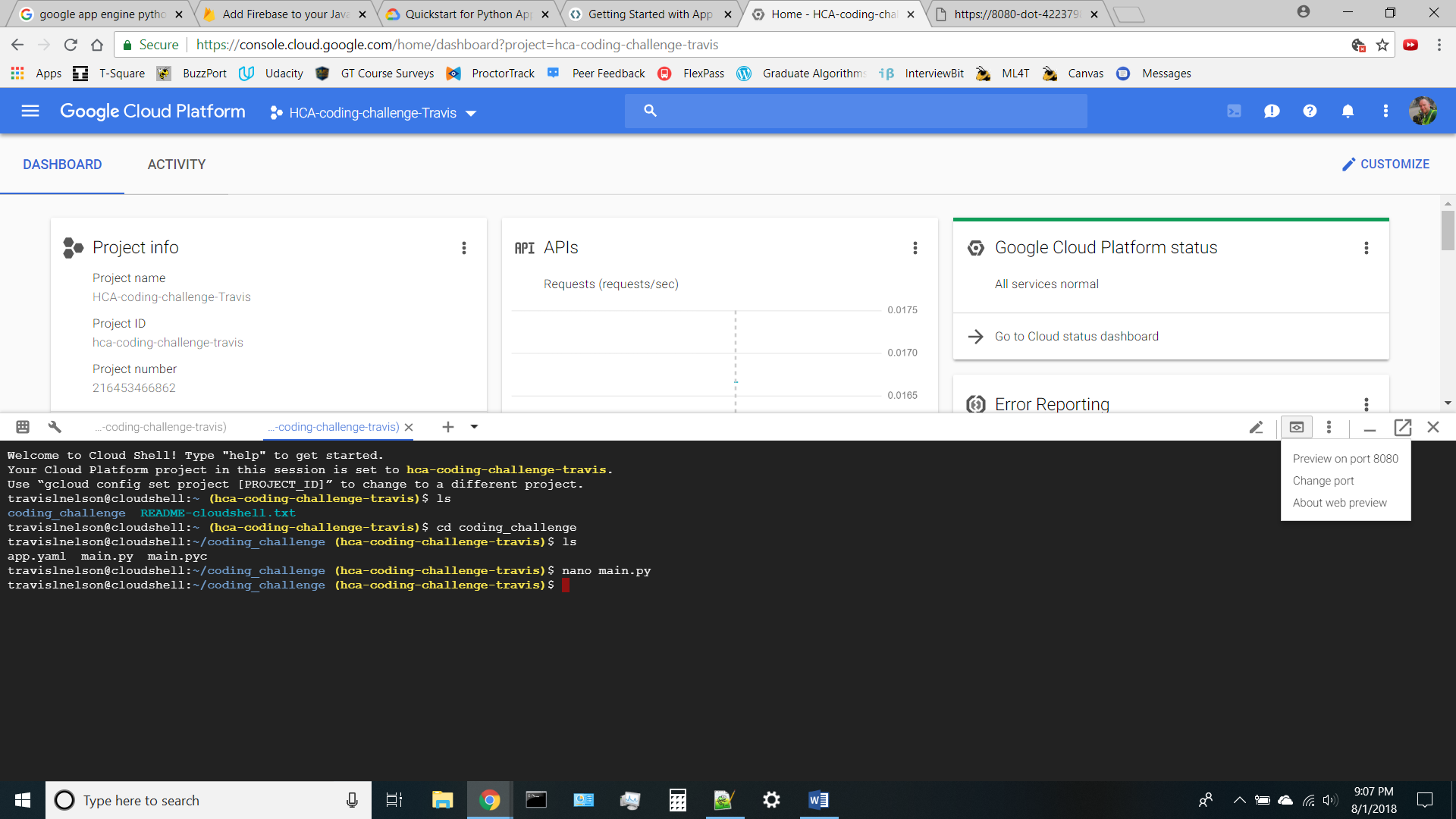


1. The shell is for a Debian-based virtual machine that already has all of the development tools. So convenient!
2. From the command line, create the files: main.app and app.yaml
   1. main.app is where we’ll put the code for the project
   2. app.yaml is needed for the webapp to work properly
   3. We’ll need some HTML files, I called mine index.html and results\_page.html
3. In app.yaml, we just need some standard code. Jinja is a library that’s used to render HTML that’s stored in a separate file.



app.yaml

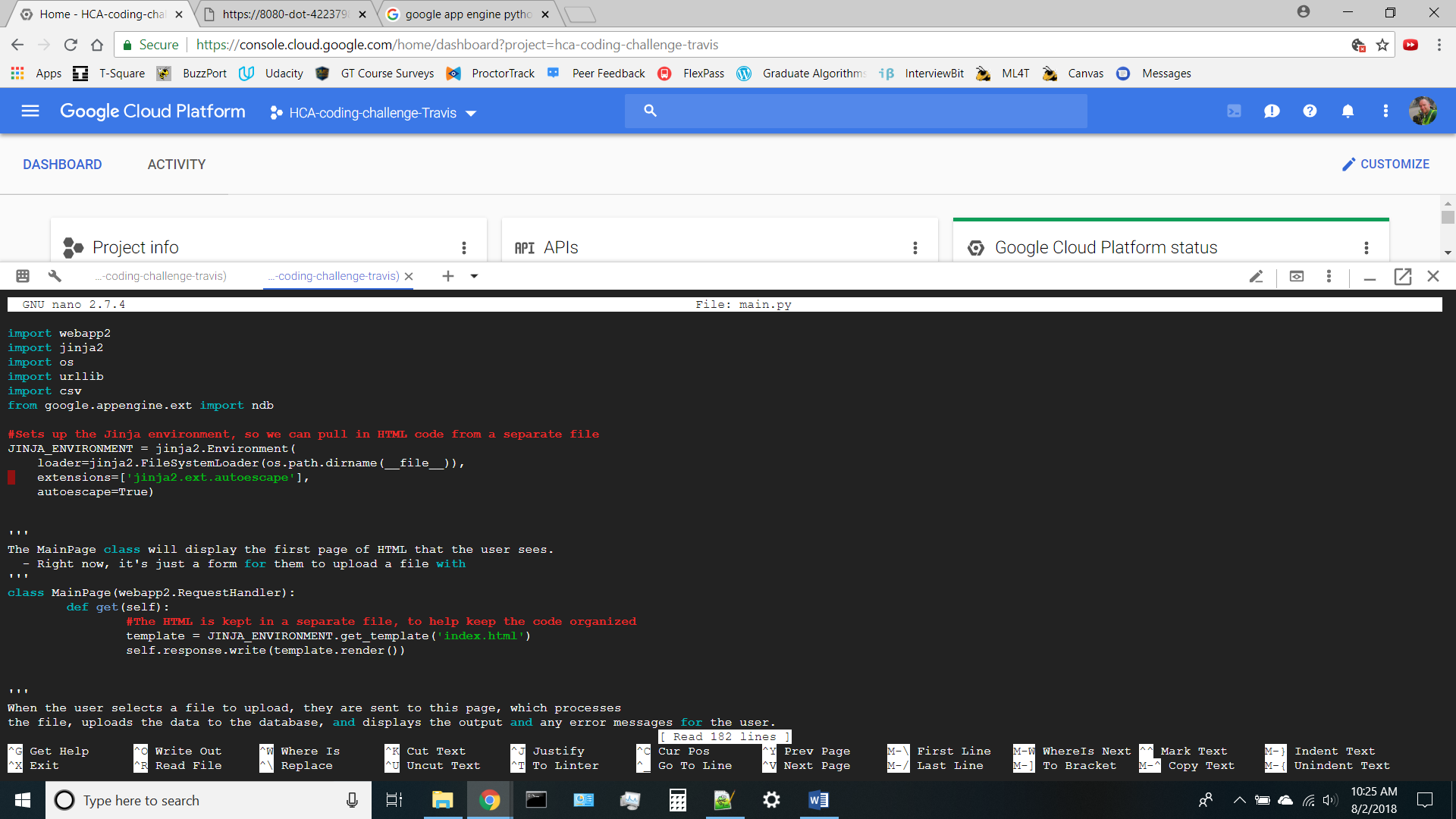
1. Eventually, we’ll need to test the code, to do that, start up a development server with the command: dev\_appserver.py ./
2. While the first command shell is running the development server, you can open a second shell to do the rest of your development activities. Click on the + to start a new shell.
3. To view your test page, click on the Web Preview button and select ‘Preview on Port 8080’:



1. To edit the files, you can use nano, vim, or emacs. There’s also a beta version of a document-style code editor, but I had problems with Python spacing using it. It might work better for other languages.

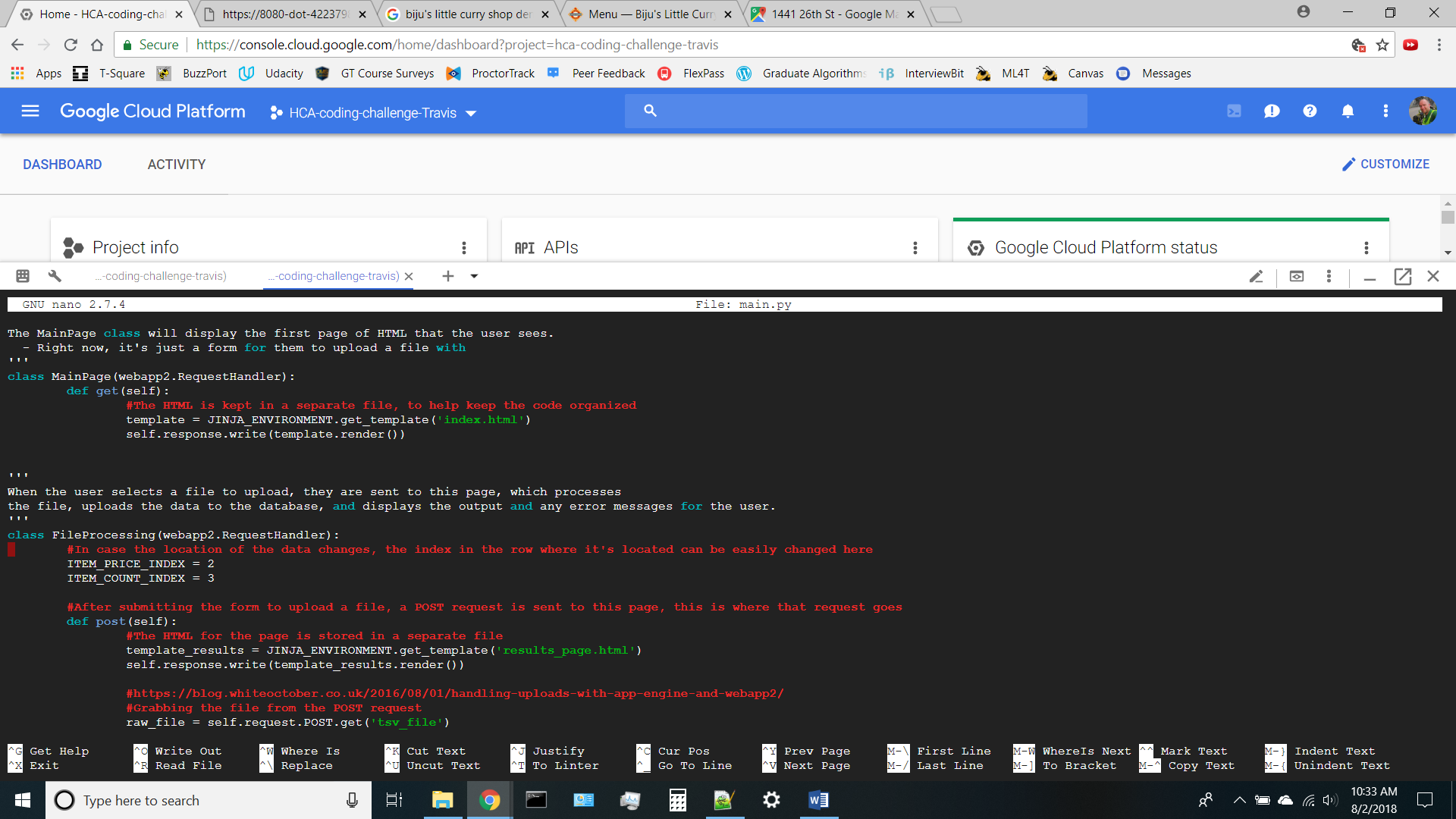
**The main.app code:**

1. First, the imports and setting up the Jinja environment:



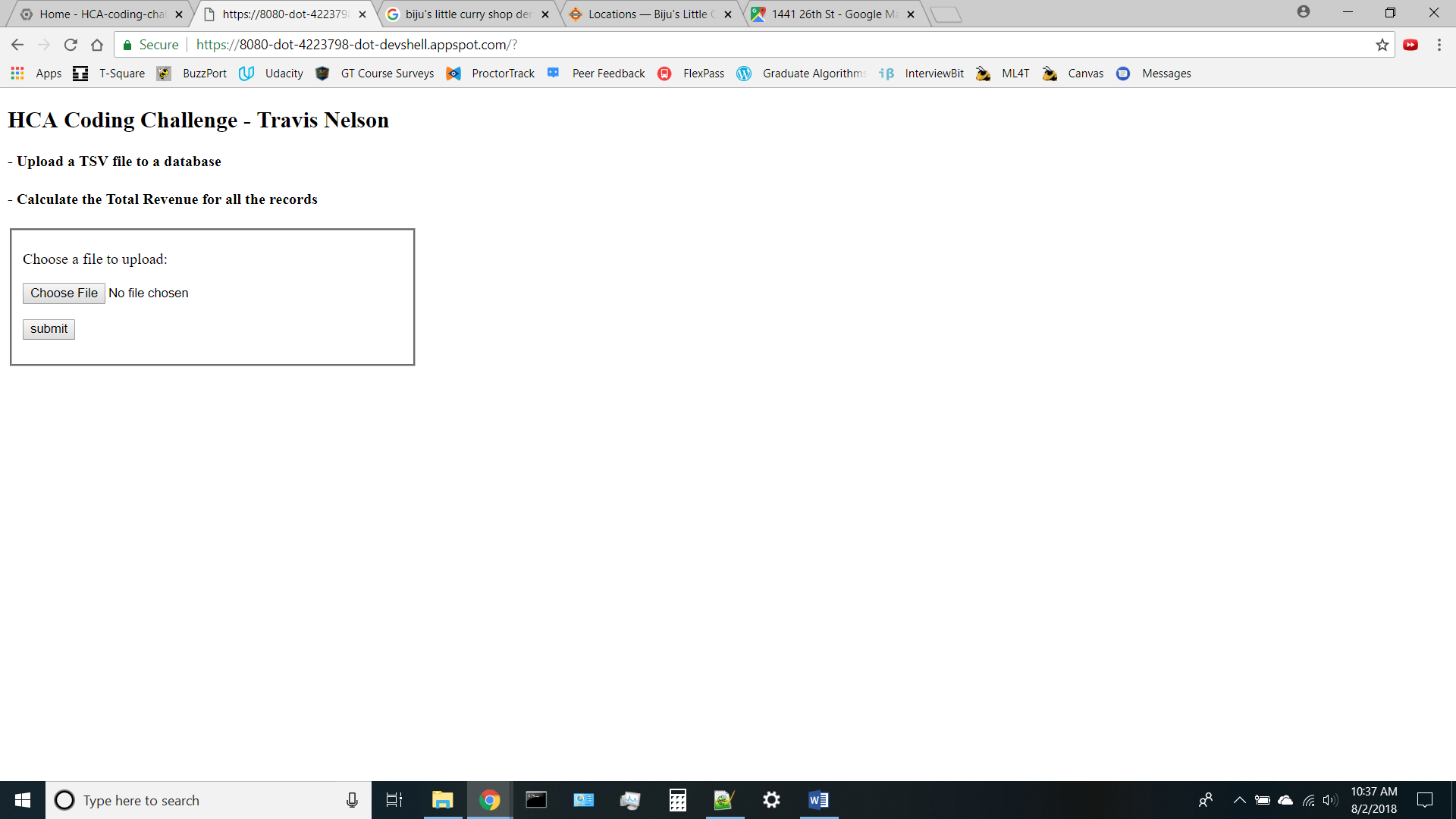
main.py

1. Each page of the Webapp is coded as a separate class, the first one is for the main page. All it does is to basically display the form for the user to upload a file. The HTML is stored in the index.html file, so the code doesn’t get too messy:



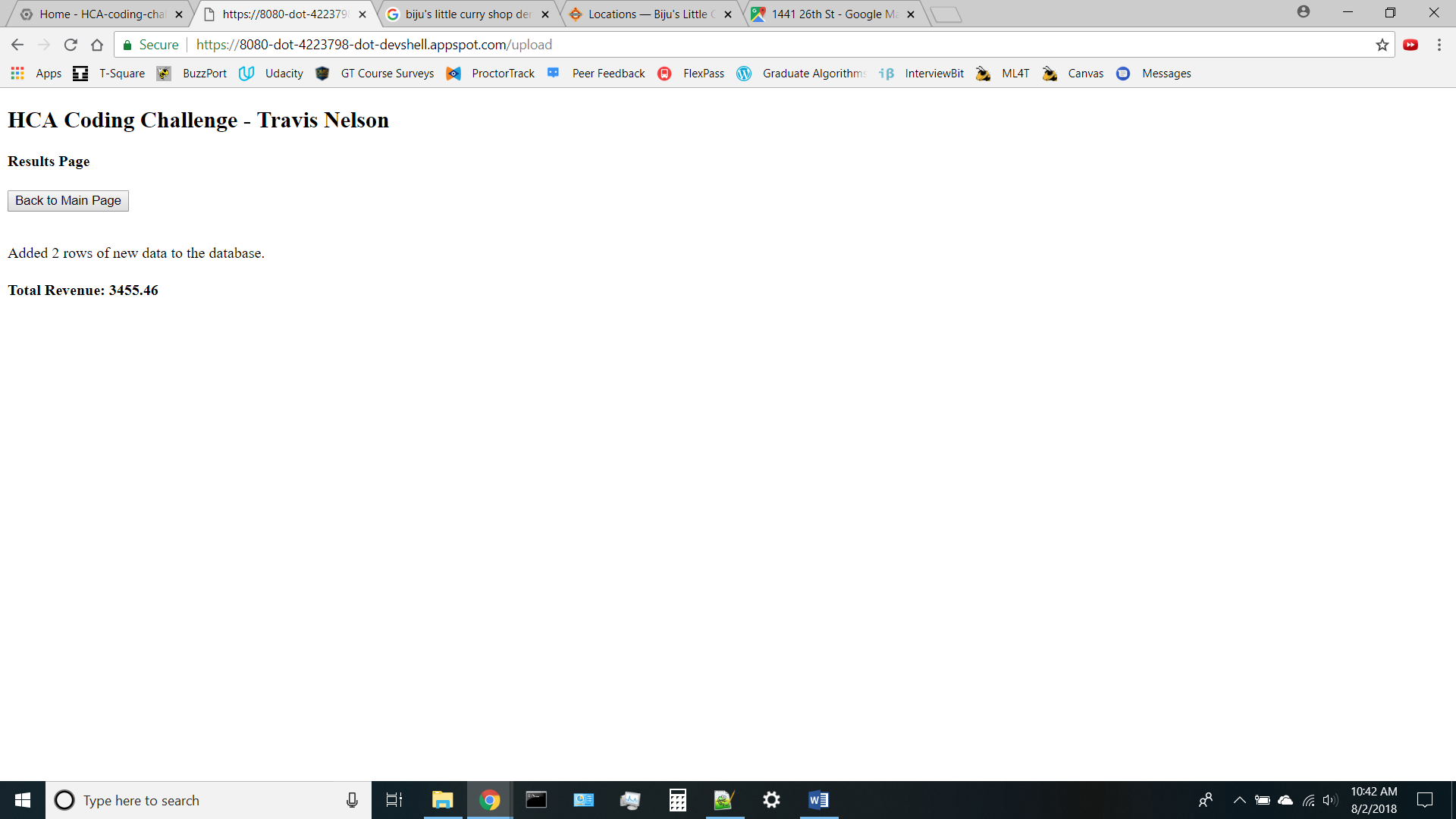
main.py

1. The first page the user sees is just the form to upload a file. Obviously it’s nothing fancy! There’s a check on the HTML form to only accept \*.tsv files.



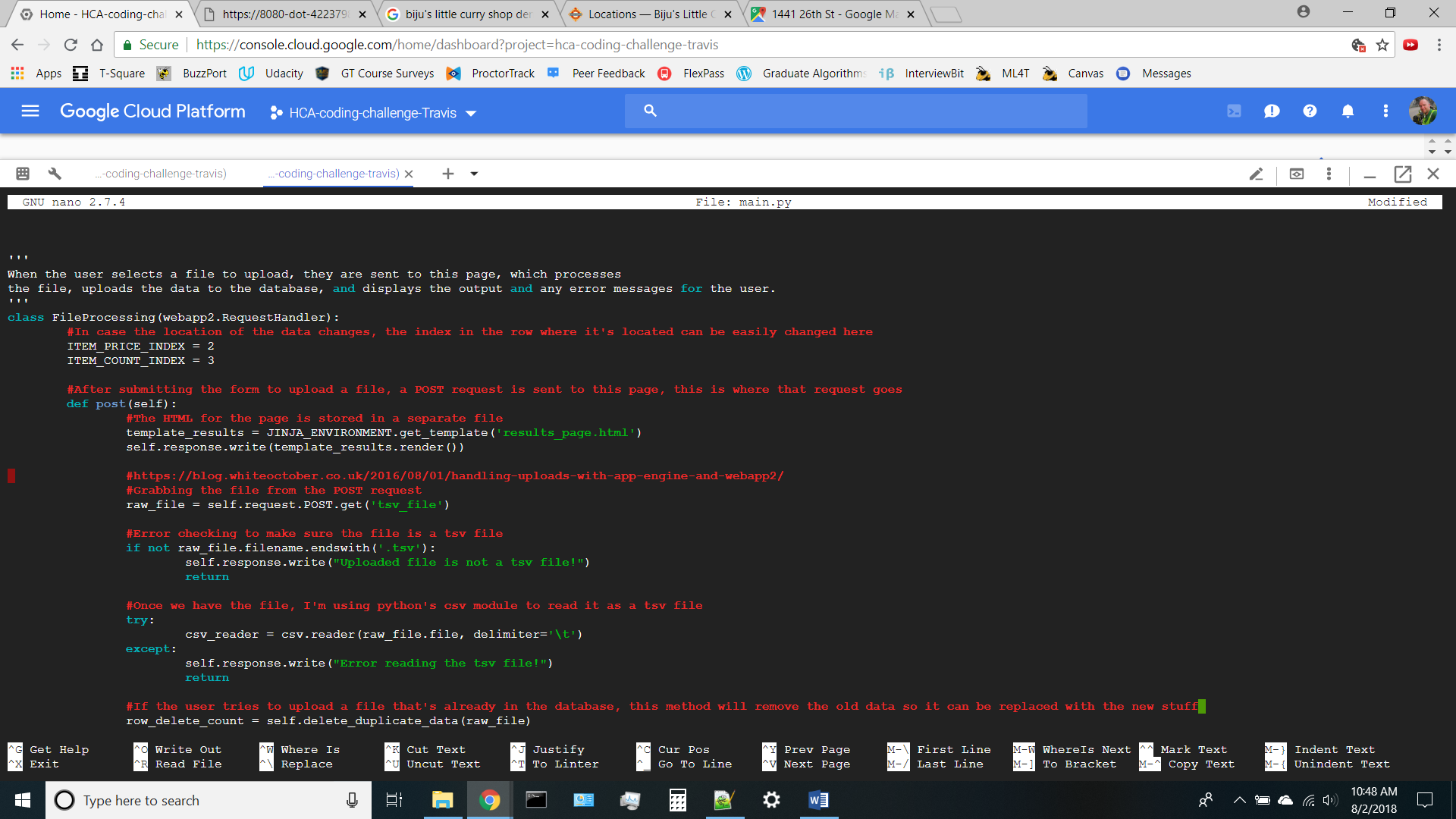
main HTML page

1. After hitting the submit button, a POST request is sent to the next page. If the user selects a file that’s already in the database, the Python code will delete the old file and all of its data and then upload the new file. Otherwise, it displays a little message telling the user how many rows of data were uploaded and the total revenue in the file, plus a little button to get back to the main page. Again, nothing fancy.



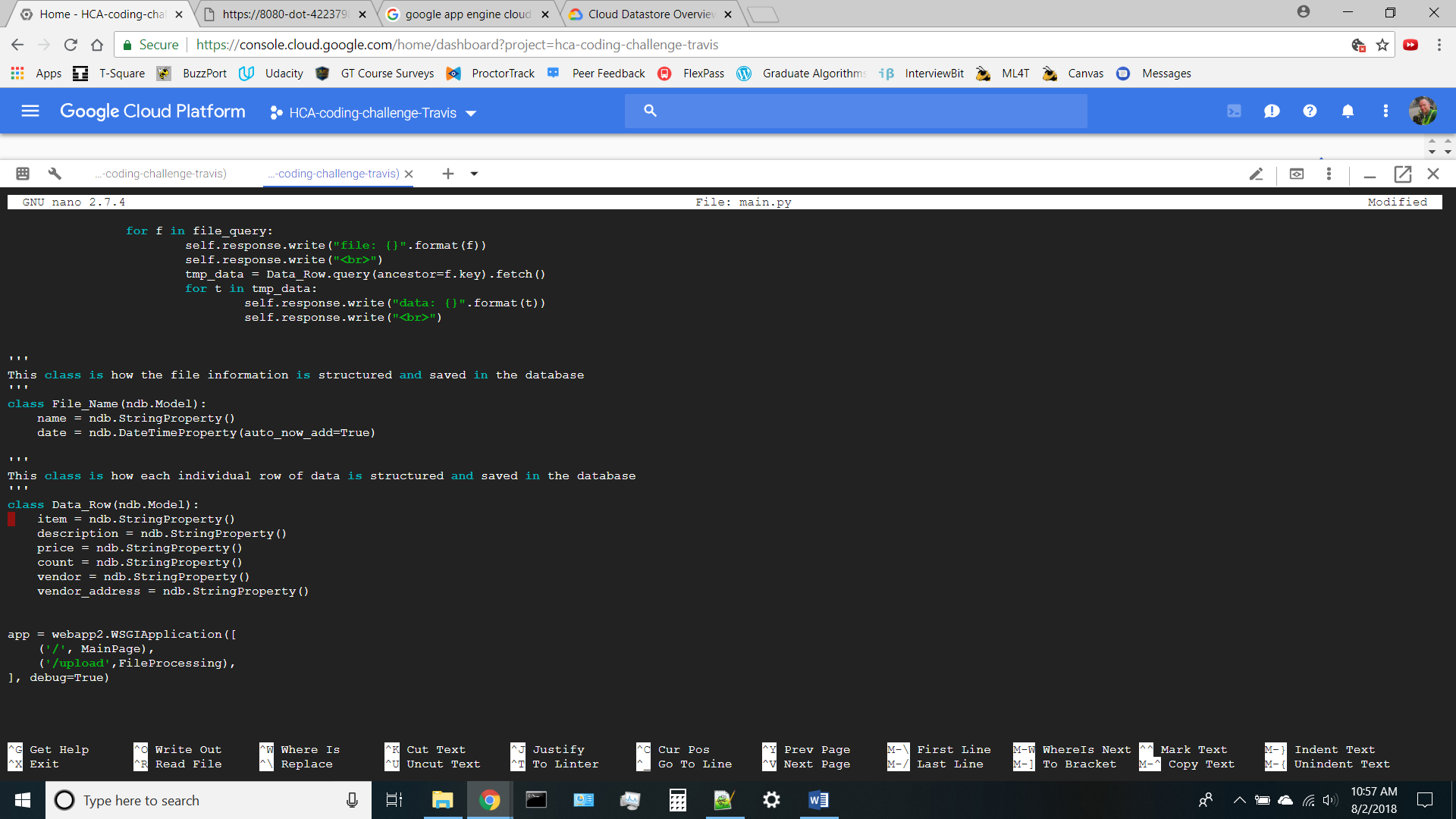
Results HTML Page

1. The code for the results page is in its own class, and the request is handled through the post method. The HTML is again stored in a separate file, then we just pull the uploaded file out of the POST request, run it through Python’s csv reader to read it as a TSV file (by specifying the delimiter as ‘\t’)



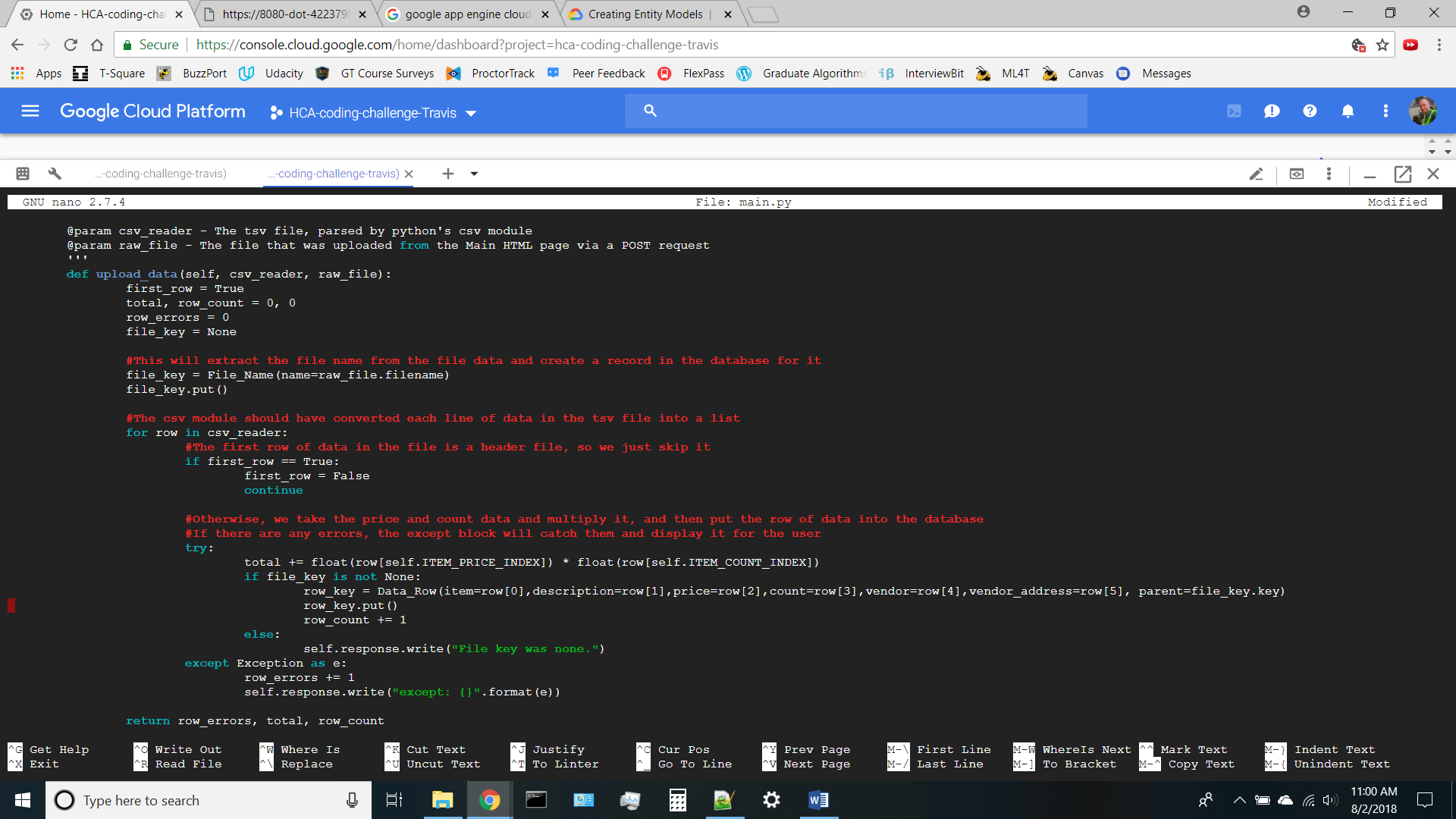
main.py

1. The data is stored in Google’s Cloud Datastore, a NoSQL document database. They call a piece of data an ‘entity’, and for this project I defined the name of the file as the parent and each row of data in the file is its child. To connect to the datastore, you use Google’s NDB client library. The data doesn’t have to be stored like this, the Datastore lets you name fields on the fly, but for structured data this works just fine. Each entity is defined by a class, and the properties of the entity are defined within the class



main.py

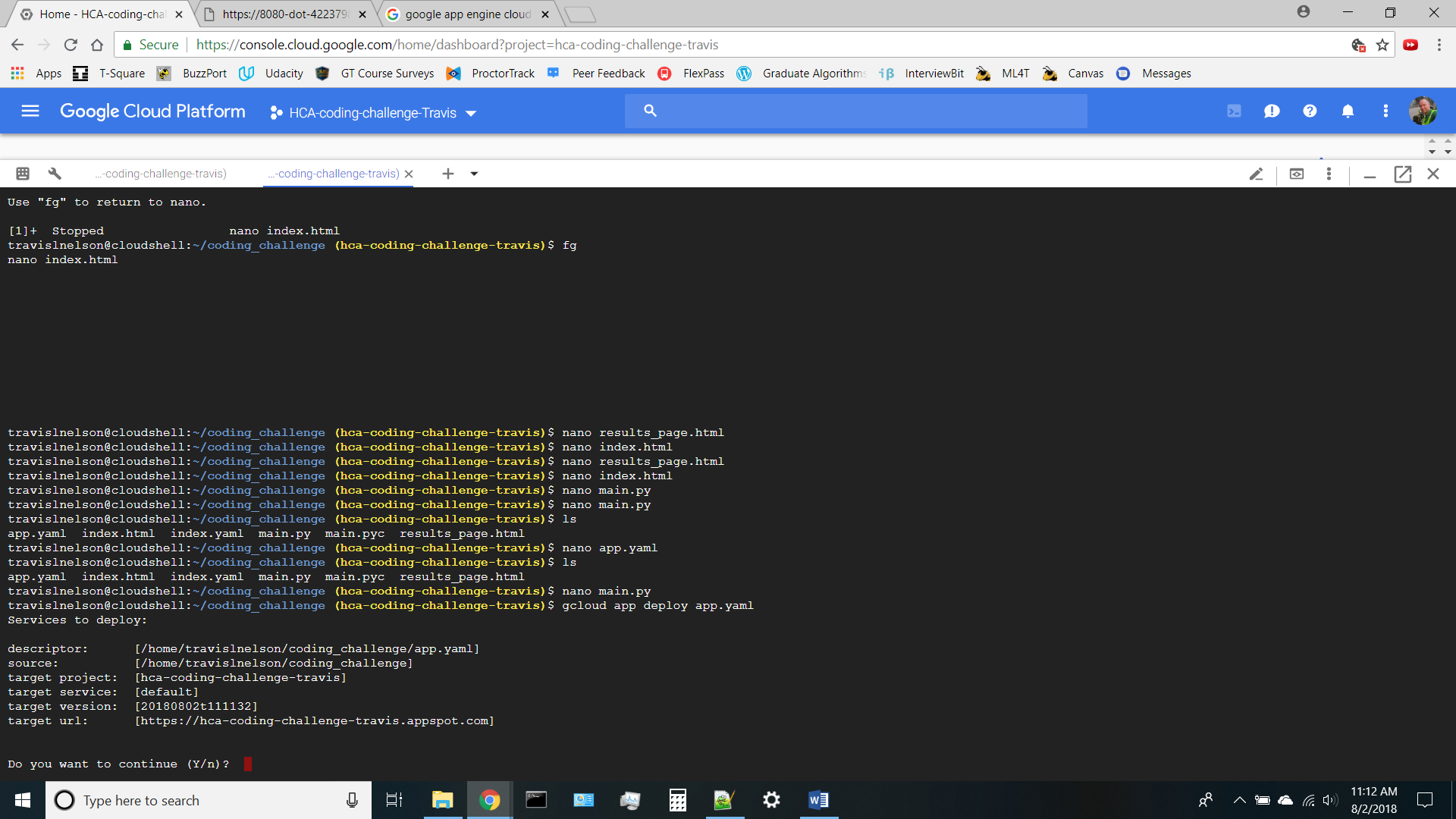
1. To add an entity to the Datastore, you create an object of whatever kind you are storing, fill out the property fields, and use .put() to save it to the Datastore, the return value is an object that contains the unique key for the item in the datastore. The parent can be specified by its key, in this case the parent was created just before this for loop, so the key was saved and used here:



main.py

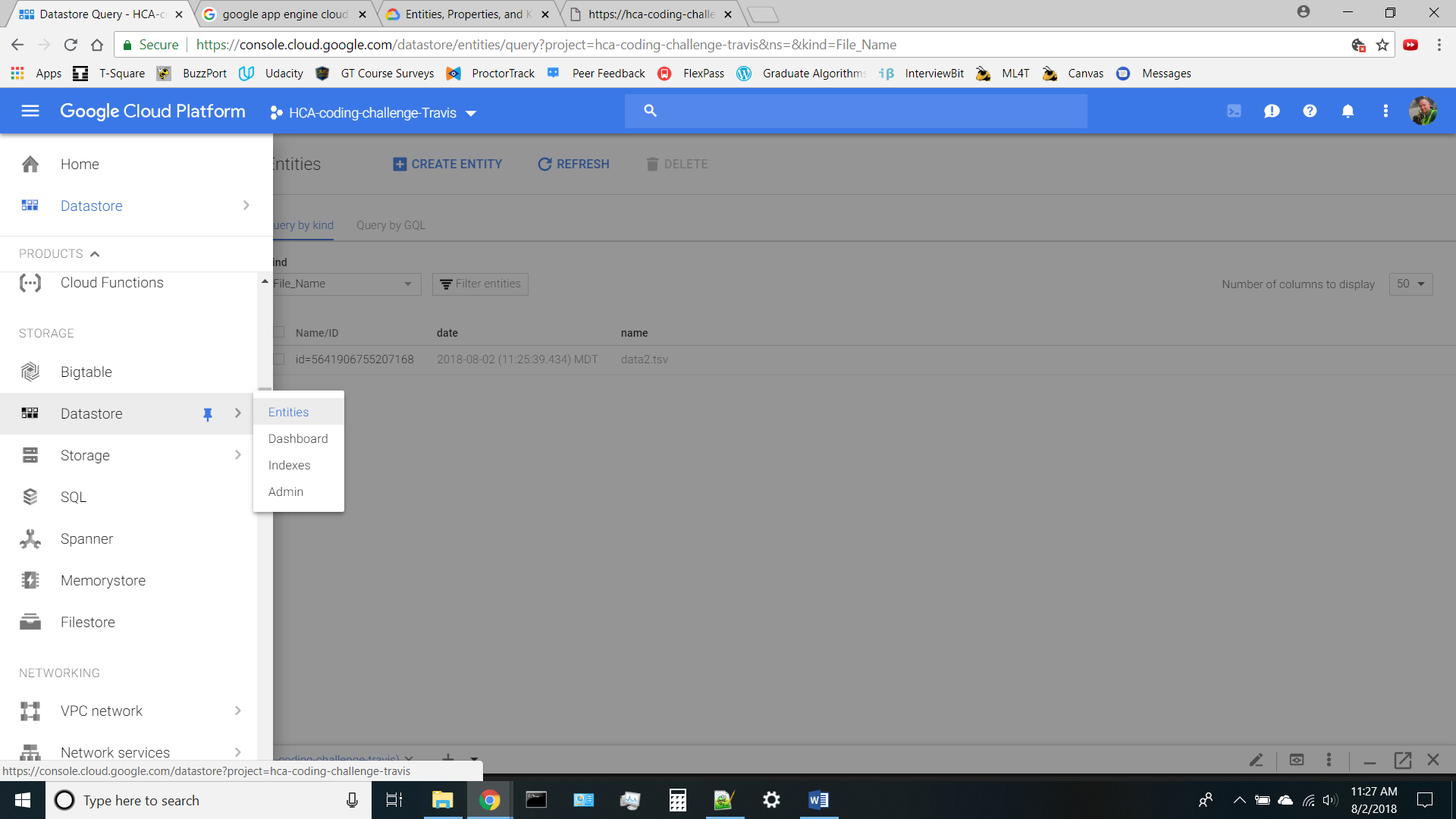
1. The self.response.write(“—“) statements just output a line of text to the HTML page
2. After code has been added or changed, the output can be easily seen by refreshing the port 8080 preview page
3. When everything is finished, deploying the app is as simple as running the following command:

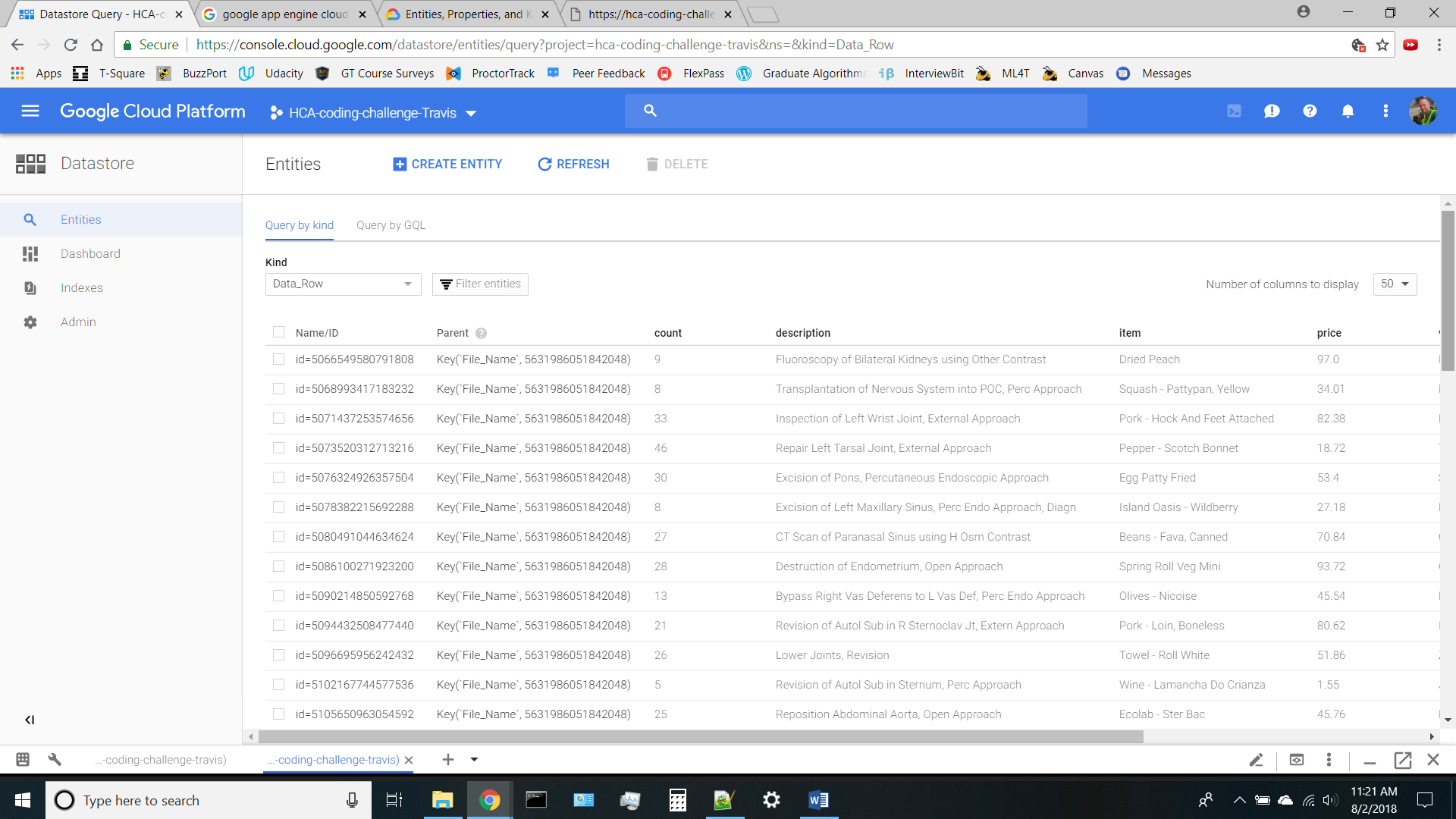
$ gcloud app deploy app.yaml



Cloud Shell

1. After it has been deployed, the data can be viewed in the Dashboard for the project. Click on the menu in the upper left hand corner, then scroll down to the Datastore





1. That’s it. Enjoy and have fun!