Instagram Replica (PaaS application) - Fotogram 1.0

This project is intended to be a basic photo sharing social media platform akin to instagram, facebook, or others. This is a fairly lightweight platform that can be expanded upon significantly to include more features.

Requirements

This project was written with Python 3.8 (and latest google-cloud-sdk and packages)

requests
gcloud
google-auth
firebase-admin
google-cloud-datastore
google-cloud-storage
google-cloud-ndb
httplib2
requests
flask
virtualenv
urllib3
protobuf

These can each be installed via pip. You must also install the Google Cloud SDK.

Database Setup

Setting up the database is straightforward. We need a firebase key, and a service account key for Google Cloud, all that is needed is to have these put into the keys folder. Rename the firebase key to firebase.json, and the service account key to fotoadmin.json.

How to Use

This website is intuitive to use, with many design cues taken from other social media websites

Sign Up

The form is divided into three inputs, username, email, and password. Usernames (and passwords) can be reused, however emails cannot be reused.

Sign In

There are only two inputs, email and password. This form also generates a cookie for session storage and passes it to the browser for further authentication purposes.

Navigation Bar

The navigation bar has the search bar, which searches users based on their usernames, this only works by the start of a username. It also has buttons for the home page, posting(+), and the profile page of the currently signed in user. This element is present on every page, including the user, profile, post, etc.

Home

The home page only contains the timeline of most recent posts by users the currently signed in user follows. Posts are sorted by latest.

Profile

The profile page can be accessed by /p/(uID) - uID denoting the user ID of a user. It has the followers amount, following amount, and post amount, along with a time line of the posts that the user posted, again sorted by latest.

Followers and Following

The followers and following pages are identical and can be accessed by /p/(uID)/followers or /p/(uID)/following), save for what they are querying. The followers queries the user followers, the following queries the user following.

Methods

@app.route('/signup', methods=['GET', 'POST'])

signup():

Either renders as default, or potentially renders after POST request, which renders an error message. This function handles sign up for the user, renders the template “signup.html”, creates a user in the firebase authentication database for authentication, then a user in the datastore database for content serving. Looks for fields in form “email”, “password”, “username”.

@app.route('/signin', methods=["GET", "POST"])
signin():

Handles signing in the user, renders the template “signin.html” and can render error message after POST request if there is any error. Accesses the firebase api through a roundabout way “signin_with_email_and_password()”. Looks for fields in form “email”, “password”. Adds a cookie to the browser that contains user info.

@app.route('/search', methods=["GET", "POST"])
search():

Queries the database for users that start with a specific string, as substrings require much much more overhead this is what I’ve found many different websites use. Serves potential users searched for.

@app.route('/', methods=['GET', 'POST'])
home():

Handles the homepage. Gets 50 last posts by users that user follows. Handles multiple post methods, one being “comment”, one being “like”, With the like button, it can either like or unlike (not shown that you can unlike). Comments are added continuously. and only five are shown on the front page, if you want to see more you must expand.

@app.route("/post", methods=['GET', 'POST'])
p():

Handles submission of files and their titles, generates post from there.

postfix():

Gets a post from the database and converts everything to be more easily readable by flask templates, returns JSon type dictionary.

@app.route('/i/<pid>', methods=["GET", "POST"])
postpage(pid):

Gets a post and renders a page with just the post in it (including image, title, author, comments, likes).

@app.route('/p/<uid>', methods=['GET', 'POST'])
profilepage(uid):

Similar to the homepage, however it just displays the profile’s posts. It also displays the profile’s posts, post amount, followers number, following number, etc via querying the keys in the profile. Displays posts similarly to the home page.

@app.route('/p/<uid>/followers')
proffollowers(uid):
Queries the user followers and displays them as a list.

@app.route('/p/<uid>/following')
proffollowing(uid):
Same as proffollowers, queries the user following and displays them as a list.

@app.errorhandler(500)
server_error(e):
Handles 500 errors.

@app.errorhandler(404)
server_error(e):
Handles 404 errors.

uploadFile(f, id):
Uploads files to current storage bucket, and returns the URL for the file.

sign_in_with_email_and_password(email, password):
Uses direct request methods to the API of the identity tool kit, normally this would be handled through having both frontend and backend separated into different servers.

raise_error(request_object):
Raises HTTP errors on request, mainly for the sign_in_with_email_and_password() return statement to verify that the JSon works.

Models:

user(ndb.Model):
Contains properties:
username (string)
email (string)
timestamp (datetime property)
followers (Key)
following (Key)
posts (Key)
ppic (String) (URL to image)

bio (String Property)

I chose this model because it allows for access to the username, email, timestamp that the user was created, the followers, and following the user has (stored in a separate object to allow for faster loading), the posts the user has posted as a separate object, the profile pic as a URL, and the bio of the user as a String.

followers(ndb.Model): - has same ID as user.

Contains properties:

num (int property, number of followers)

f (String array of uID's)

I chose this model because it allows for access to the total number of followers as an integer (for much faster viewing, instead of fetching and counting the total amount of followers). Along with a list that contains the IDs of all those that are following, again for easy querying.

following(ndb.Model): - has same ID as user

Contains properties:

num (int property, number following)

f (String array of uID)

I chose this model because it allows for access to the total number of followings as an integer (for much faster viewing, instead of fetching and counting the total amount of followings). Along with a list that contains the IDs of all those that are following, again for easy querying.

uposts(ndb.Model):

Contains properties:

num (int property, number of user posts)

f (String array of postIDs)

I chose this model because it allows for easy access to all of the posts a user makes with IDs being able to be packaged into a single query of IDs.

post(ndb.Model):

Contains properties:

title (String)

timestamp (datetime)

author (String) - uID

pic (String) - URL to image

likes (Key)

comments (Key)

I chose this model because it allows for easy access to all of the components and a relatively small (initial) query size for minimal data egress. The post only contains the title, timestamp, author (as uID, not as key), link to pic, the likes as a key(to make it so that relatively little is accessed through the post, and comments the same.

likes(ndb.Model):

Contains properties:

likecount (Integer)

likers (String Array) - uIDs

I chose this model because it allows for easy querying of individual comments by ID, instead of by Key, for easy access to the uIDs used (so it's easier to like and unlike), instead of a structured property. Along with a likecount number so the number of likes is easily displayed.

comment(ndb.Model):

Contains properties:

author (String) - uID

text (String) - comment body

timestamp (DateTime)

I chose this model because it allows for easy access to individual comments. It is somewhat easier to query ID strings rather than Keys so that was why it was chosen to access. The text is chosen because there needs to be some way to access the comment body. The timestamp was chosen solely for sorting purposes.

comments(ndb.Model):

Contains properties:

num (int) - number of comments

comments (Key) - comment Key.

I chose this model because it allows for easy querying of individual comments by Key, instead of a structured property. Along with a number of comments for easy tracking.