Building and Deploying a Login System backend using Flask, SQLAlchemy and Heroku

with a Heroku Postgres Database

Aakanksha NS Follow
May 17 · 7 min read

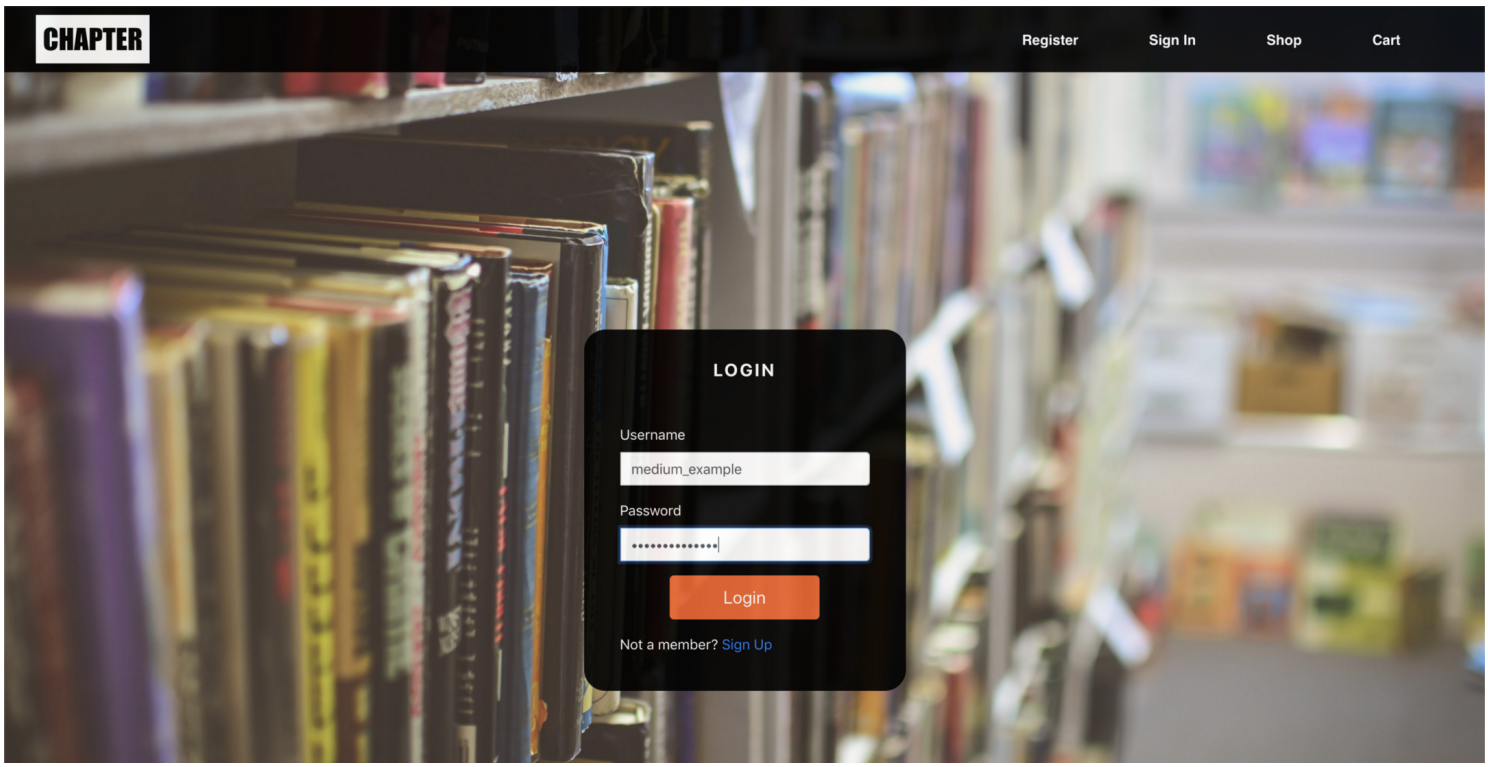


Image by author

Almost all websites these days have a login system, where you can register if you're a new user or sign in if you're an existing one. There can be a lot of moving pieces if you're trying to build this system from scratch for your application. This article gives a step-by-step overview to create and deploy REST APIs for a login system but can be extended to any REST API in general.

All the code related to this article can be found in the following repo:

aakanksha-ns/flask-heroku-login

APIs written in Flask using a Heroku Postgres database to register a user and log into account. Deployed on Heroku ...

github.com

1) Create a Heroku account and create a new application

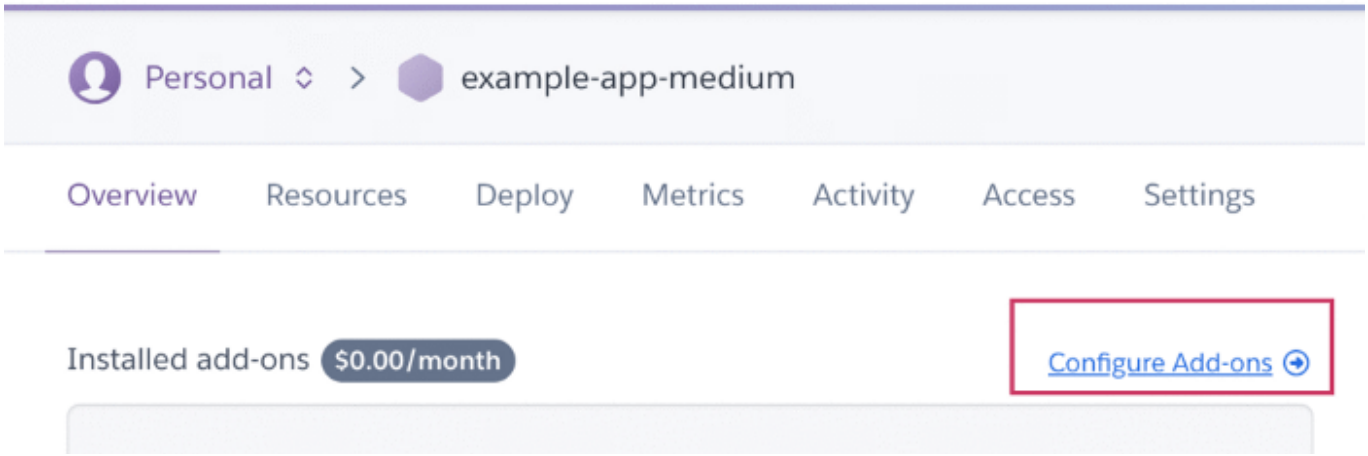
The first thing you'd need to do is create an account on Heroku if you don't already have one.

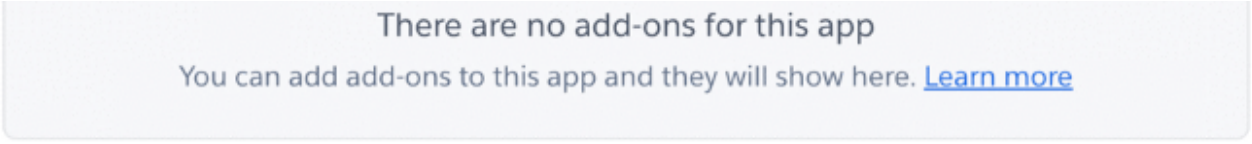
Sign in to your account and select the `Create new app` option to create a new Heroku application.

Dashboard->New->Create new app (Image by author)

2) Add Heroku Postgres to application

Now you need to add Postgres to the application. Navigate to the page of the application you just created, select the `Configure Add-ons` option, search for `Postgres` and provision it.





Dashboard->'your application'->Configure Add-ons (Image by author)

3) Install Heroku CLI, PostgreSQL, and PopSQL

Heroku CLI

To be able to interact with Heroku from the command line, you'd have to install Heroku CLI using the following command (assuming you have Homebrew already installed):

```
$ brew tap heroku/brew && brew install heroku
```

PostgreSQL, and PopSQL

Why we need them: You could create your database's tables from your Flask app. In my implementation, however, I've created the database and tables locally using PostgreSQL and pushed that to Heroku because I found it more straightforward creating tables using a normal SQL editor (I used PopSQL) as compared to SQLAlachemy. I've used Flask and SQLAlchemy only to add elements to the database or retrieve elements from it.

Install Postgres: `brew install postgres`

Start PostgreSQL server: `pg_ctl -D /usr/local/var/postgres start`

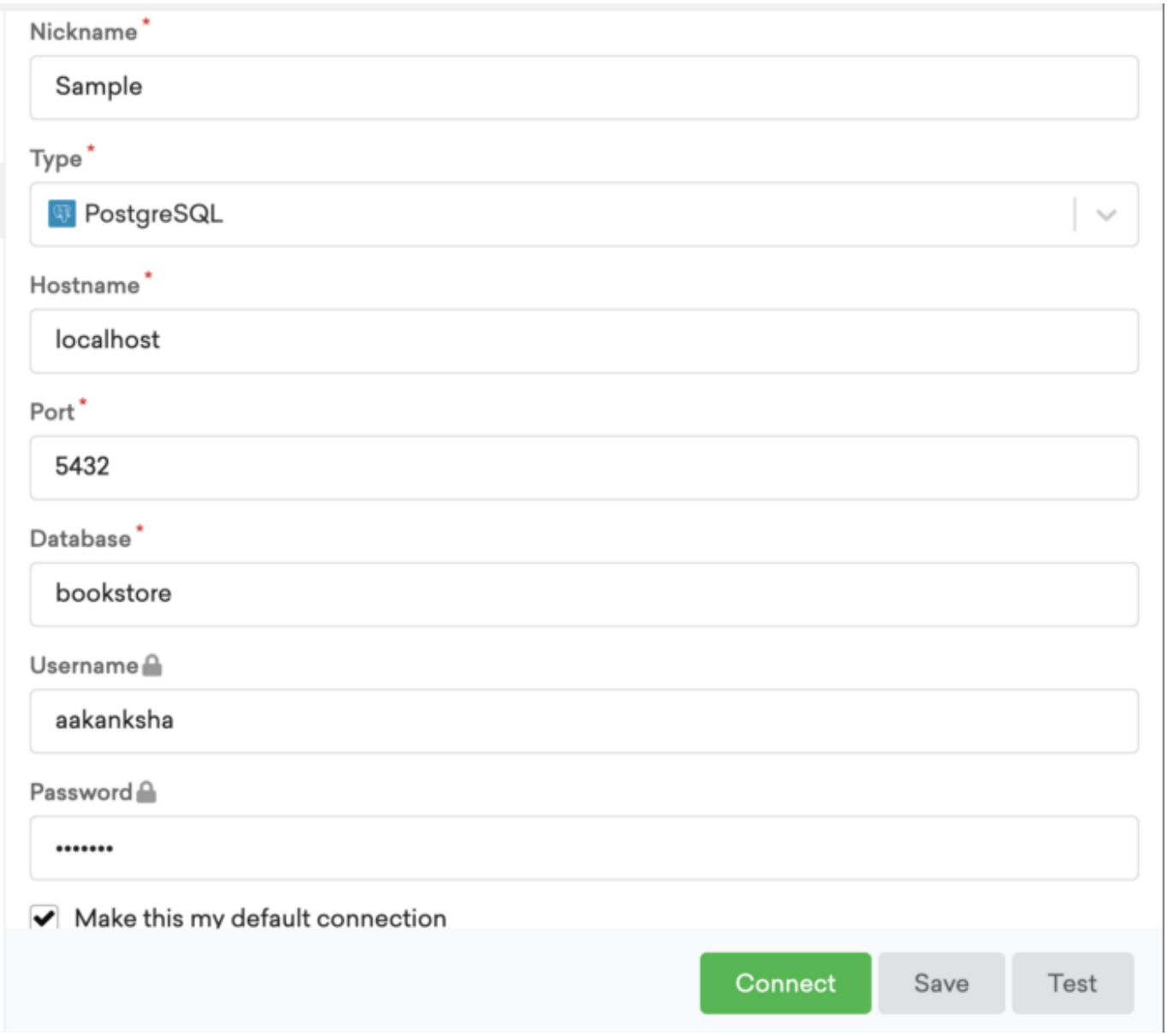
Creating database and user: Postgres sets up a set of default users but they are super user accounts — they can do anything, including delete databases. It's best to create new users with access only to a particular database:

```
$ psql postgres

CREATE ROLE aakanksha WITH LOGIN PASSWORD 'blah';
CREATE DATABASE bookstore;
GRANT ALL PRIVILEGES ON DATABASE bookstore TO aakanksha;
```

Install PopSQL: PopSQL is a very popular and easy to use SQL editor. Download the application from https://popsql.com/ and install it.

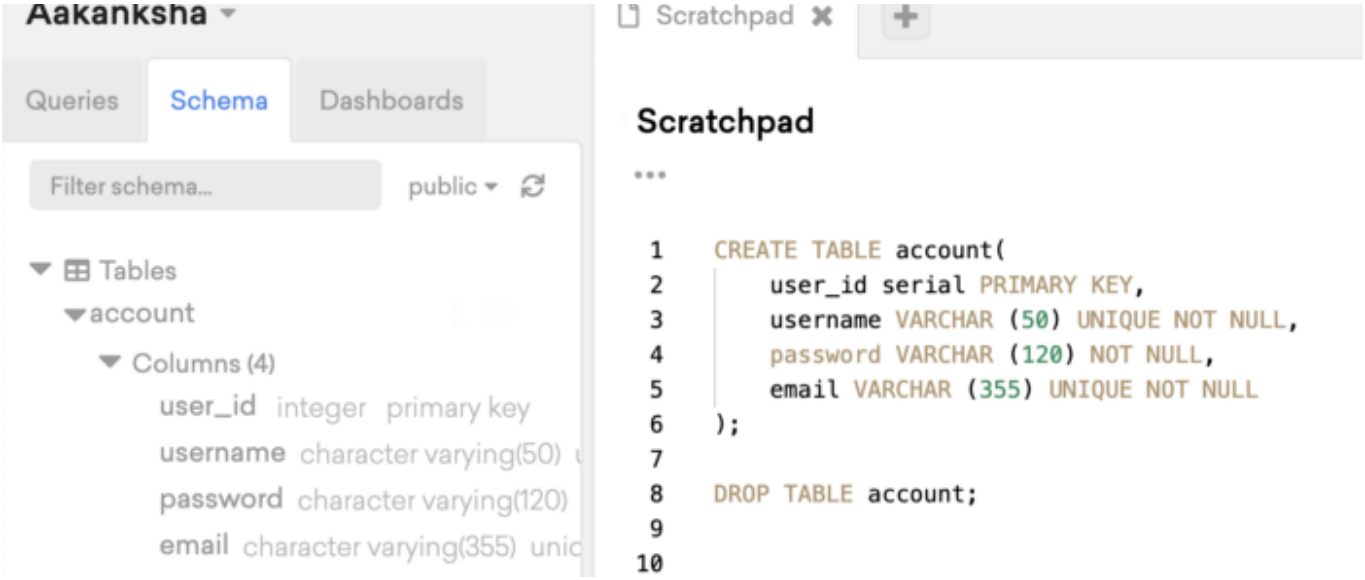
Connect PopSQL to database: You'll have to provide details like `port number` (postgres uses 5432 by default), `database name` etc. for PopSQL to be able to access your PostgreSQL database. You can just enter the following details:



(Image by author)

4) Create Tables and push Database to Heroku

Create Tables on the PopSQL scratchpad:



Execute the 'DROP TABLE' command only if you made a mistake while creating (Image by author)

Push to Heroku:

```
$ heroku pg:push bookstore DATABASE --app example-app-medium
```

5) Install required libraries

Now that the database is set up, we can move on to the API part. Make sure you install all the required libraries. You can just store the following list of libraries I've used into a text file (`requirements.txt`) and execute the command following it:

```
certifi==2020.4.5.1
click==7.1.1
Flask==1.1.2
flask-heroku==0.1.9
Flask-SQLAlchemy==2.4.1
itsdangerous==1.1.0
Jinja2==2.11.1
MarkupSafe==1.1.1
psycopg2==2.8.5
SQLAlchemy==1.3.16
Werkzeug==1.0.1
gunicorn==19.5.0
flask_cors
```

```
pip install -r requirements.txt
```

6) Set up environment variable to access Heroku Database

Since you have to access the remote database on Heroku through your flask app, to establish a connection to it, you'd need the database URL. Because the remote database's URL can change due to many reasons, It is best practice to always fetch the database URL config var from the corresponding Heroku app when your application starts. You can do this by setting an environment variable.

Since this URL is particular to a particular project which is limited to a particular directory, it wouldn't make sense to add it to your global environment variables. This is where direnv comes in. It is an environment switcher that allows to load/unload environment variables depending on the present working directory.

To set the variable for your Flask application you'd have to add the following command to a new file called `.envrc` and place it in the root directory of your project:

```
export DATABASE_URL=$(heroku config:get DATABASE_URL -a example-app-
medium)
```

After creating the `.envrc` file, run `$ direnv allow` to activate the variables.

5) Setup Flask app config

Here's some basic config you need in your `config.py` file in the root directory:

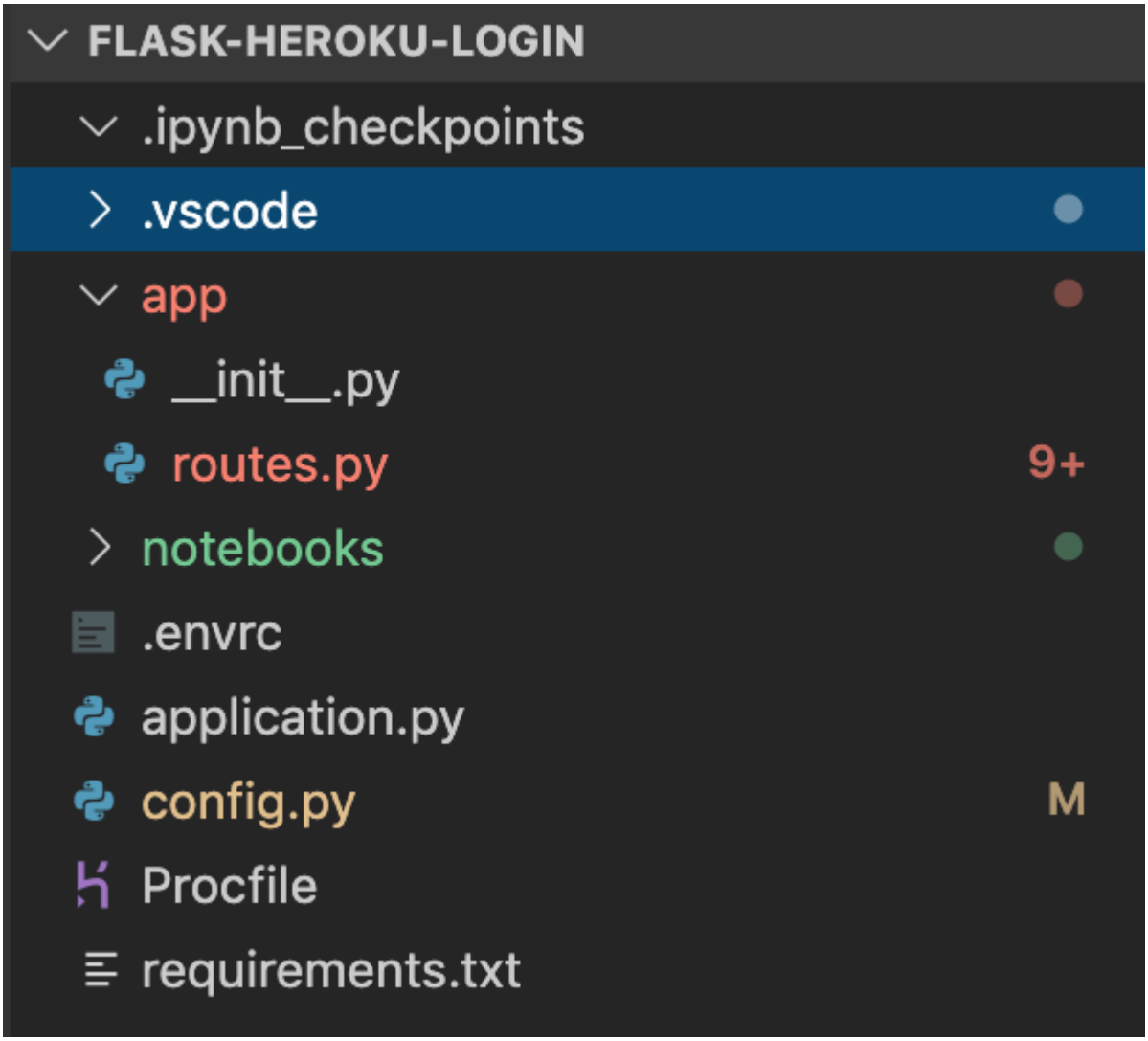
```
import os

class Config(object):

SQLALCHEMY_TRACK_MODIFICATIONS = True
SECRET_KEY = os.urandom(24)
SQLALCHEMY_DATABASE_URI = os.environ['DATABASE_URL']
```

6) Setup application.py and app directory

Here's what my folder structure looks like:



(Image by author)

Flask names an app 'application' by default, so you'd need an `application.py` file for the app to run. You can change the name of your app if you want to (using `$ export FLASK_APP = 'your file name'.py`)

I like putting all Flask related code into a subdirectory called `app` instead of the root.

My `application.py` looks like:

```
from app import application
```

And within the app subdirectory, here's my `__init__.py` (initialization script that marks directories on disk as Python package directories):

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import create_engine

# Initialization
application = Flask(__name__)
application.config.from_object(Config)

DB_URI = application.config['SQLALCHEMY_DATABASE_URI']
engine = create_engine(DB_URI)

from app import routes
```

The API code lies in the `routes.py` file.

7) Create API and test locally

To see the full `routes.py` code, follow this link:

https://github.com/aakanksha-ns/flask-heroku-login/blob/master/app/routes.py

For a login system, you'd need two methods:

Register

This method creates an entry in the `account` table for a new user. You can do an additional check to make sure the `username` or `email` does not already exist in the table.

Since it's not safe to store the raw password in a database due to security concerns, you can generate a password hash using the `werkzeug` library and store the hash instead of the real password.

```
@application.route('/register', methods=["GET", "POST"])
def register():
    username = request.args.get('username')
    email = request.args.get('email')
    password = request.args.get('password')
    password_hash = generate_password_hash(password)
    account = Table('account', metadata, autoload=True)
    engine.execute(account.insert(), username=username,
                   email=email, password=password_hash)
```

```
10      return jsonify({'user_added': True})
```

register.py hosted with ❤ by **GitHub** **view raw**

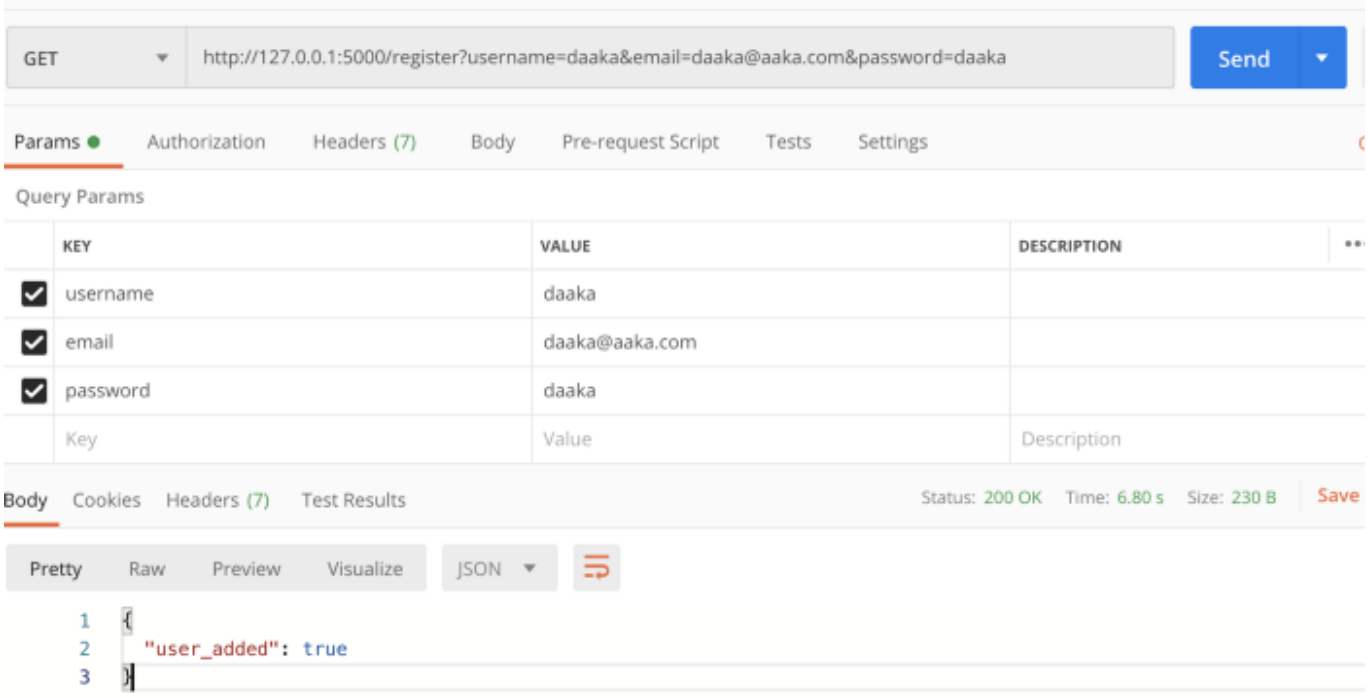
Sign In

This method just checks if the password entered matches with the original password for the given username

```
1  @application.route('/sign_in', methods=["GET", "POST"])
2  def sign_in():
3      username_entered = request.args.get('username')
4      password_entered = request.args.get('password')
5      user = session.query(Accounts).filter(or_(Accounts.username == username_entered, Acco
6                                            ).first()
7      if user is not None and check_password_hash(user.password, password_entered):
8          return jsonify({'signed_in': True})
9      return jsonify({'signed_in': False})
```

flask_sign_in.py hosted with ❤ by **GitHub** **view raw**

Once done, you can use `$ flask run` to start the flask server and test your API using Postman:



(Image by author)

8) Create Procfile

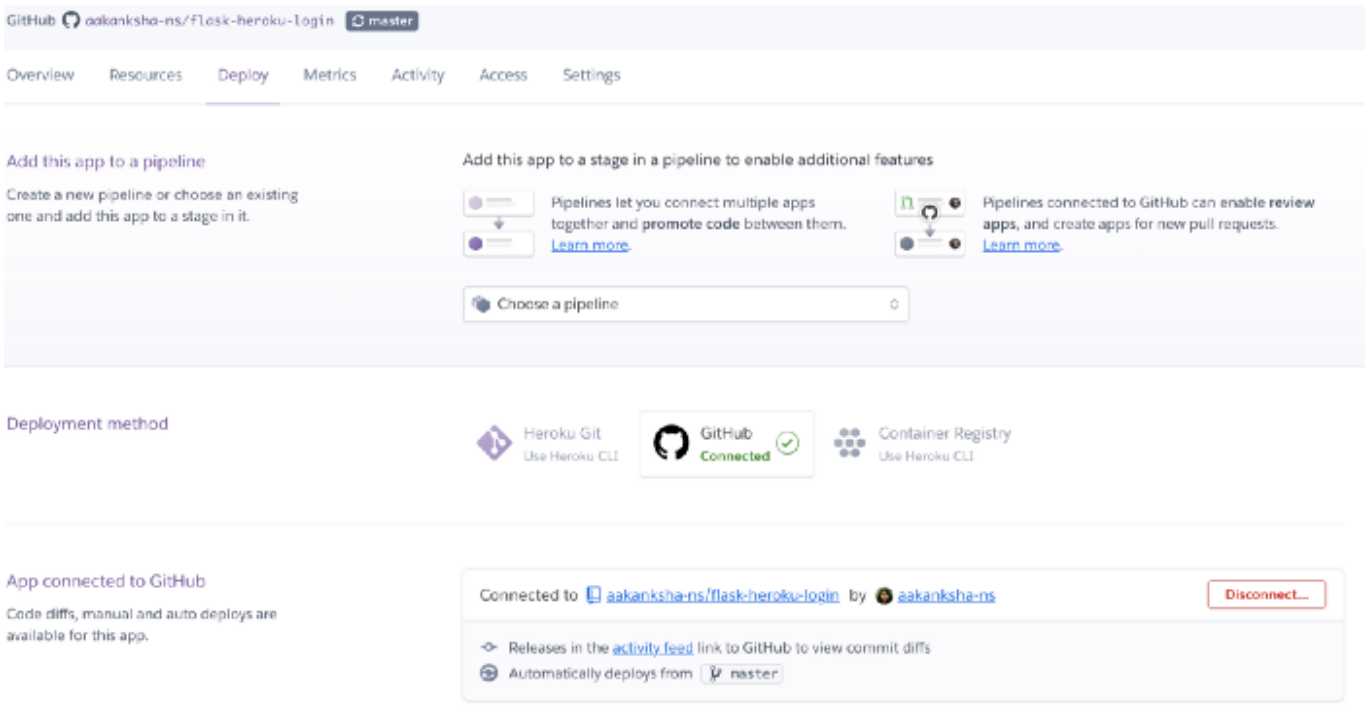
Procfile is a mechanism for declaring what commands are run by your application's dynos on the Heroku platform.

You can use `gunicorn` that takes care of running multiple instances of your web application, making sure they are healthy and restart them as needed, distributing incoming requests across those instances and communicate with the web server.

```
web: gunicorn application:application
```

9) Push code to Github Repo, connect repo to Heroku app

To deploy the APIs you just created, the easiest way is to push your code to Github repository and connect this repo to your Heroku application:

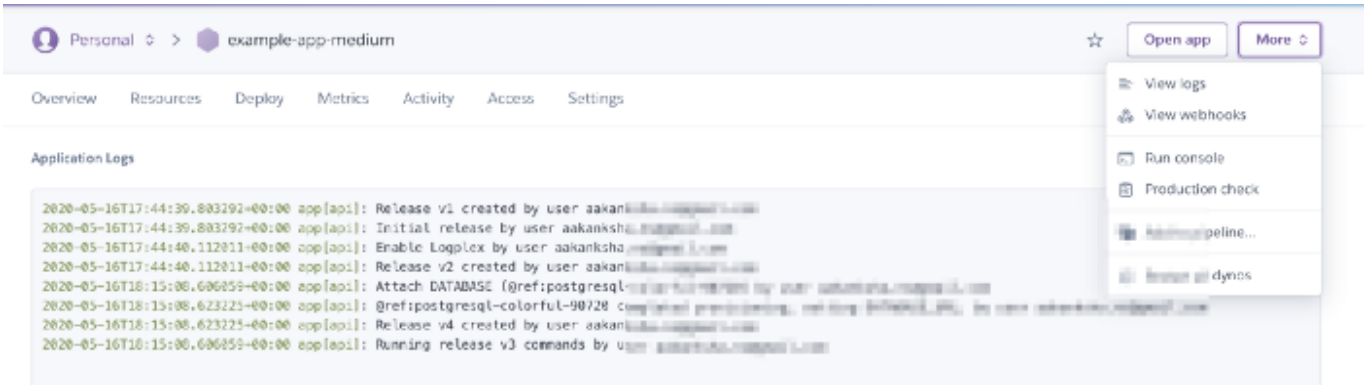


(Image by author)

9) Deploy Flask app

You can set up an automatic deployment or deploy the Flask app manually from the Heroku dashboard. If there are any errors during deployment, they'll show up in the

build log. Once deployed, you can hit the API using Postman and see that it's working as expected. If there's an error after deployment, you can view the log this way:



(Image by author)

10) Create UI

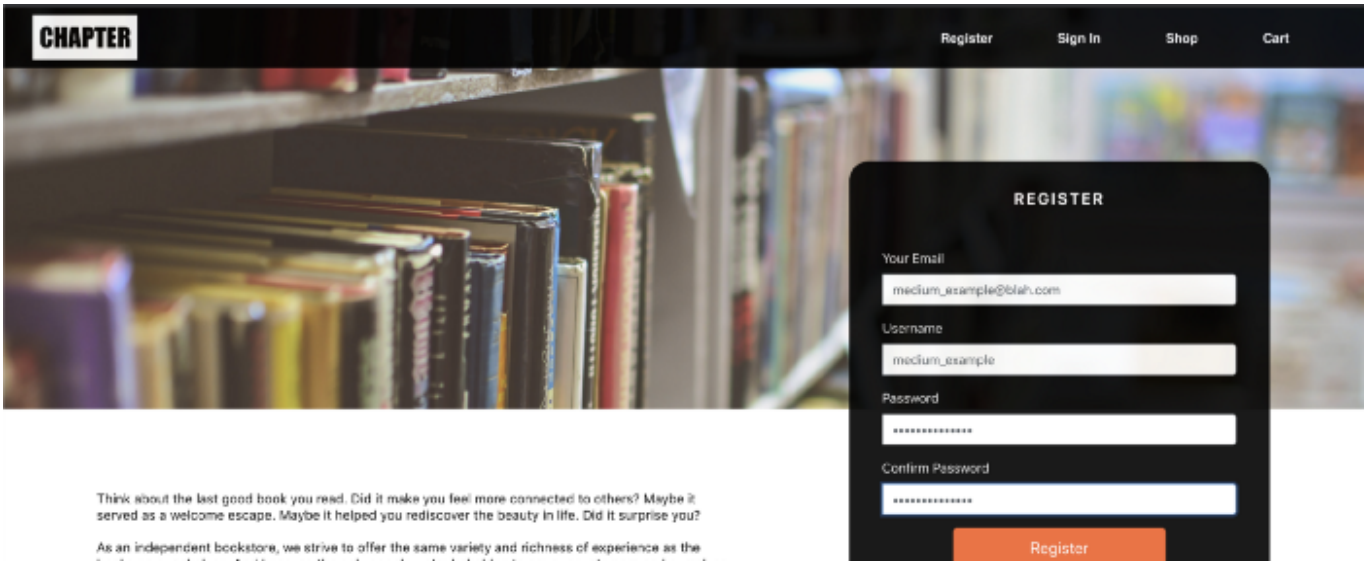
In case you're interested in creating a frontend for your login system, you can check out my React and Redux based UI:

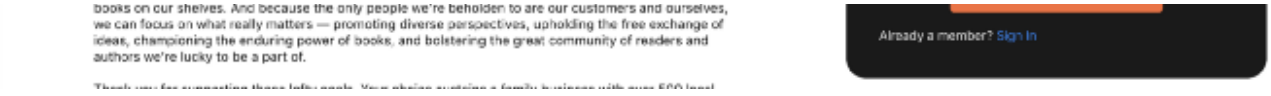
aakanksha-ns/book-store

This project was bootstrapped with Create React App. In the project directory, you can run: Runs the app in the...

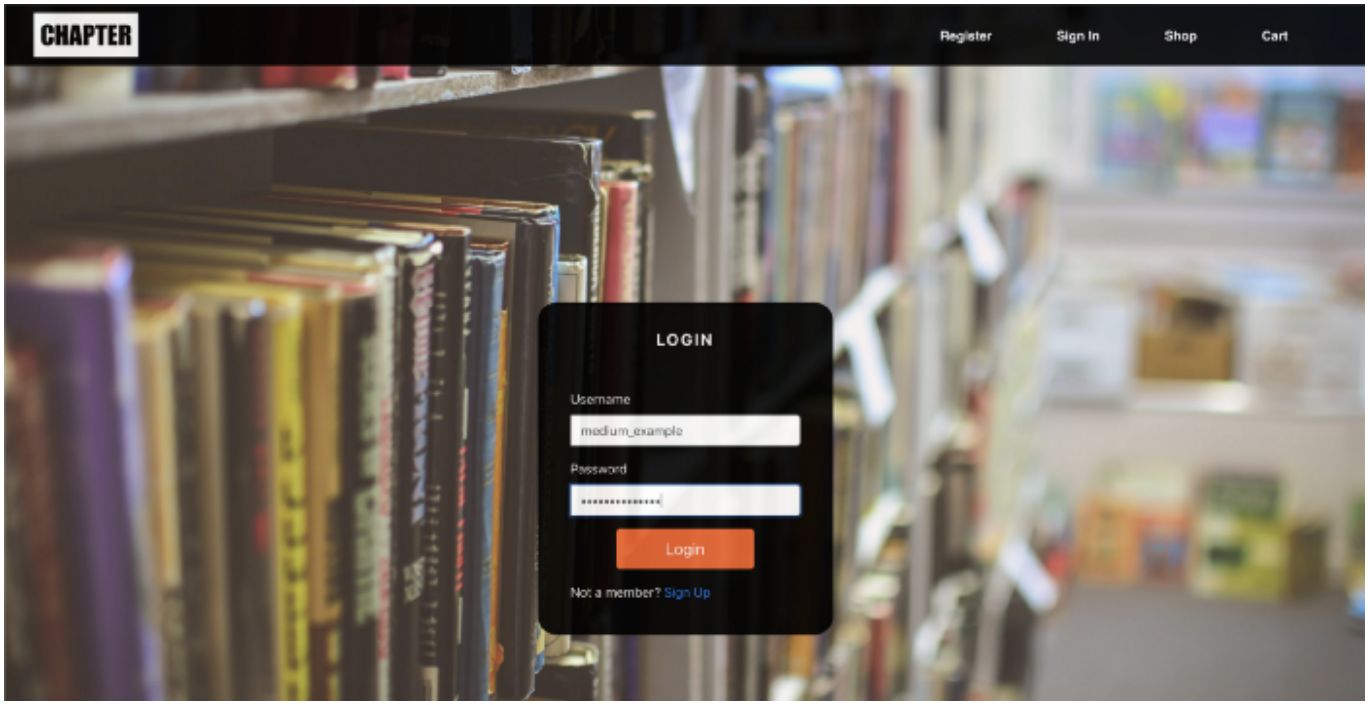
github.com

Here are a few screenshots of the application I've built:

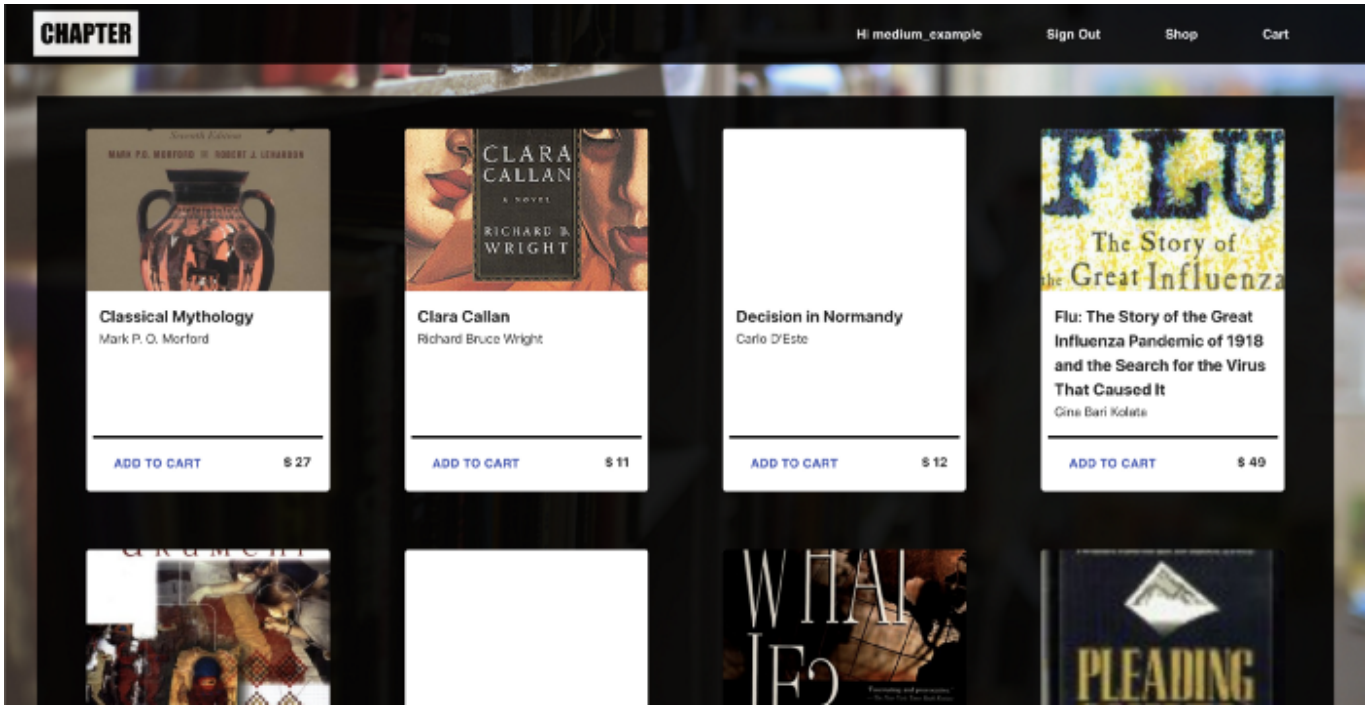




(Image by author)



(Image by author)



(Image by author)

References:

- https://thecodersblog.com/PostgreSQL-PostGIS-installation/
- https://www.codementor.io/@engineerapart/getting-started-with-postgresql-on-mac-osx-are8jcopb

Programming Sql Database Flask API