Nate Pierce
CIS525 Term Project Proposal
Dr. Jinhua Guo
University of Michigan-Dearborn

Project Title and Team

Project title: CampSite, consisting of sole team member Nate Pierce.

Web Link

https://campsite-app-a13844a96212.herokuapp.com/
Give the instance some time to spool up, they are on the cheapest instance type which takes some time to start working.

Test credentials: piercen@umich.edu / test123

Proposal

Introduction

This article proposes the purpose, design, and implementation roadmap for a web application intended to give users the ability to pin locations of campsites they visit on public land, giving the campsite a rating, uploading photos, and provide other metadata for the campsite that would be useful for their own tracking purposes or for sharing and discovering from other users. The target audience are RV campers, off-road camping, hunters, backpackers, hikers, kayakers, etc. The name of the app will be called Camp Site.

Competitors include: Campendium, Campgrounds.RVLife, TheDyrt, CampsitePhotos

The problem this application intends to solve is to provide a medium for outdoors enthusiasts to easily save and share information about places anyone can stay at during their outdoor recreation. This work is motivated by my own desire (and, from discussions I've had with other enthusiasts) for such an application.

Modules

Map and GPS Pins

Users can scroll through satellite imagery (Google Maps-esque) to add or view pins for campsites.

Campsites

Campsite objects are the core component of the app. Campsites will be searchable by keywords and by viewing the geospatial map. Campsite metadata will consist of:

- Location (lat, lon, altitude)
- Road access (yes or no)
- Gallery (photos)
- Potable water access (wells, pumps, fountains, etc. as yes or no)
- Natural water access (ponds, lakes, streams, etc. as yes or no)
- Type (public campground, backcountry campground, dispersed campsite, etc.)
- Visibility: public vs private
    - Publicly visible sites can be seen by any other use
    - Privately listed sites are only listed for the user who added them
- Possibly: a rating e.g. out of five stars?

User Accounts

Users will be able to make a list for campsites, add campsites, search for campsites. Users will have their own basic profiles, including:

- Name
- Location

Operations

A non-exhaustive list of CRUD operations relating to the web app.

1. Create user account requiring email, password, nickname.
2. Guests can log in to an existing account.
3. Add campsites to app using form, with either public or private visibility.
    a. Add photos, location, and other metadata
4. Registered Users can delete maps (only ones that they have personally added). Non-registered users may view only publicly listed campsites.
5. Allow users to create and delete lists associated with their account (possibly shareable) that are user created lists of campsites.
6. Users can view other users, user lists, and campsites.
7. Campsites searchable from both the map and from a search using filters.

User Roles & Access Control

- Guest (unregistered user): view map, and pinned publicly listed camp sites. Can not add sites or edit existing ones. Guests have no user profile. Essentially, read only operations.
- Registered users: may add campsites, or edit campsites that the specific user has added. Users have their own unique profile which they may edit (they may only edit their own profile). Mostly read only operations, with limited writing capability (on objects that the user owns).
- Administrators: Administrators have their own profile, and can add, modify, or delete any user and their associated profile. They may also add or delete any private or publicly listed campsite, and edit campsites from any user, including other administrators. Administrators will also have control over some title level configurations (exact details to be ironed out in implementation)

Team Leader

This subsection will be updated as work progresses -- work for the next week should be planned out as the project gets implemented, keeping the dates list below in mind.

1. Team Leader is Nate Pierce (sole team member) will complete all work.

- Week of October 2nd. Decide on:
  - Database: SQLite or other
  - Back-end: Python (Flask or Django) or other
  - Front end: React, JS
  - Hosting service: AWS, Heroku, Linode
  - Create git repo, add boilerplate code, relevant directories, etc.
- Week of October 9th.
  - Final decisions on frontend, backend, RDBMS, hosts, etc.
  - Make git repo w/ boilerplate Flask code, ensure server runs locally.
- Week of October 16th:
  - Begin construction of core views (login, signup, home, profile), including base html to extend with Jinja.
  - Add form for adding campsite.
  - Barebones DB implementation -- add tables and columns.
- Week of October 23rd:
  - Finish authentication views: signup, and login.
  - Add model for User.
  - Write logic for authentication, leaning on Flask.
- Week of October 30th:
  - Deploy app to host (Heroku), make app publicly accessibly.
  - Move from sqlite3 (development) to postgresql (production).
  - Ensure app runs on Heroku, parity with local development (add environment variables for flask and database).
- Week of November 6th:
  - Begin work on core modules: map view, pins on map.
  - Integrate User profile fields into DB for profile view.

Important Dates

Proposal due: October 1, 2023
Progress report due: November 14, 2023
Final presentation and demonstration: December 5, 2023
Final report due: December 10, 2023
Spreadsheet: Here

From Operations to Pages

The root URI is the home page. A dynamic navbar displays on all web pages. Depending on whether the user is logged in or not, different elements/navbar items will be shown. For instance, an authenticated user can view the campsite form and list dropdown menu, as well as the “My Profile” navbar item.

The main view is the home page, which displays:

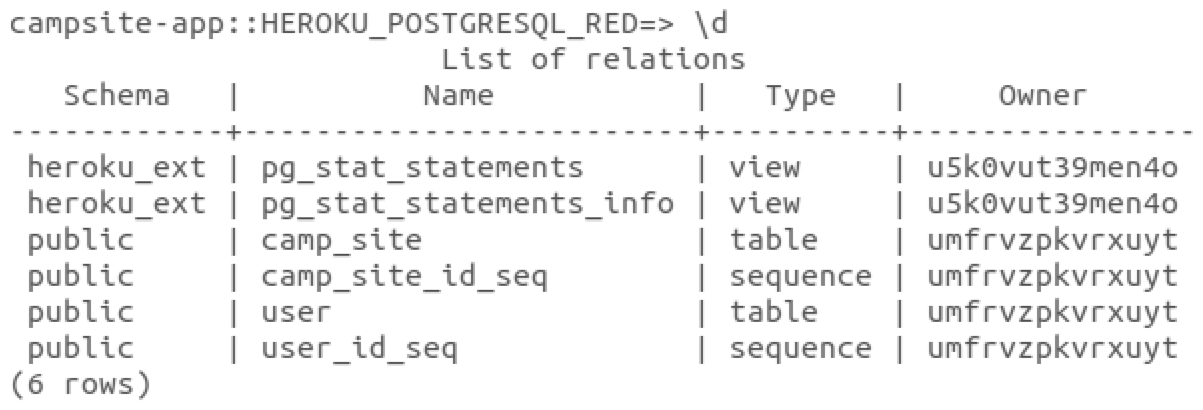
- OSV Map with pinned campsite locations, which the user can select for more details.
- Form to add campsites to the map or a user list.

Other views include:

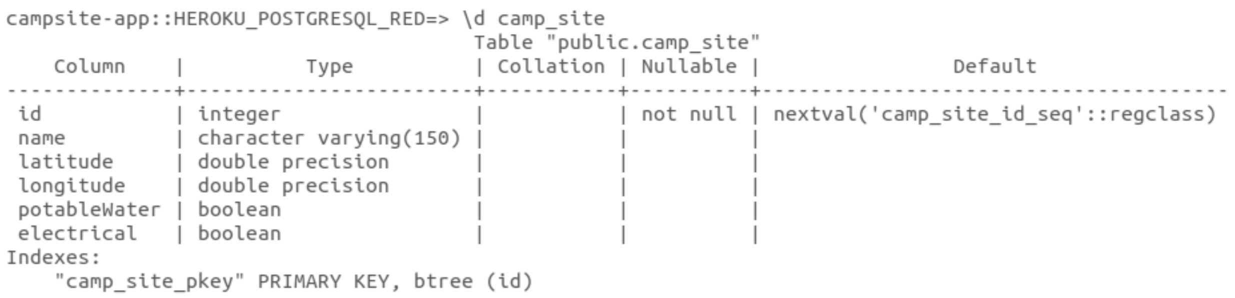
- Profile page, where users can see basic information about the selected user.
- Sign-up page and log-in page.

Database Design Schema

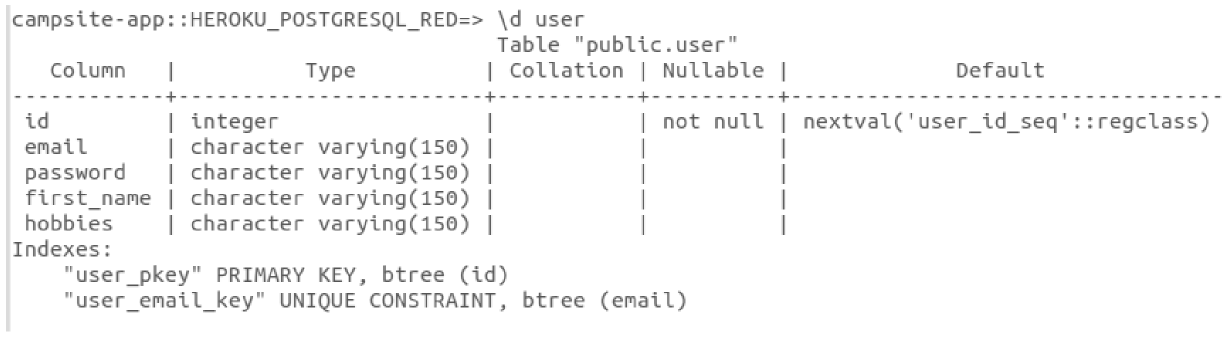
The current schema has the structure:



```
campsite-app::HEROKU_POSTGRESQL_RED=> \d
                       List of relations
   Schema   |          Name           |   Type   |     Owner
------------+-------------------------+----------+----------------
 heroku_ext | pg_stat_statements      | view     | u5k0vut39men4o
 heroku_ext | pg_stat_statements_info | view     | u5k0vut39men4o
 public     | camp_site               | table    | umfrvzpkvrxuyt
 public     | camp_site_id_seq        | sequence | umfrvzpkvrxuyt
 public     | user                    | table    | umfrvzpkvrxuyt
 public     | user_id_seq             | sequence | umfrvzpkvrxuyt
(6 rows)
```



```
campsite-app::HEROKU_POSTGRESQL_RED=> \d camp_site
                                      Table "public.camp_site"
    Column    |          Type          | Collation | Nullable |                Default
--------------+------------------------+-----------+----------+---------------------------------------
 id           | integer                |           | not null | nextval('camp_site_id_seq'::regclass)
 name         | character varying(150) |           |          |
 latitude     | double precision       |           |          |
 longitude    | double precision       |           |          |
 potableWater | boolean                |           |          |
 electrical   | boolean                |           |          |
Indexes:
    "camp_site_pkey" PRIMARY KEY, btree (id)
```

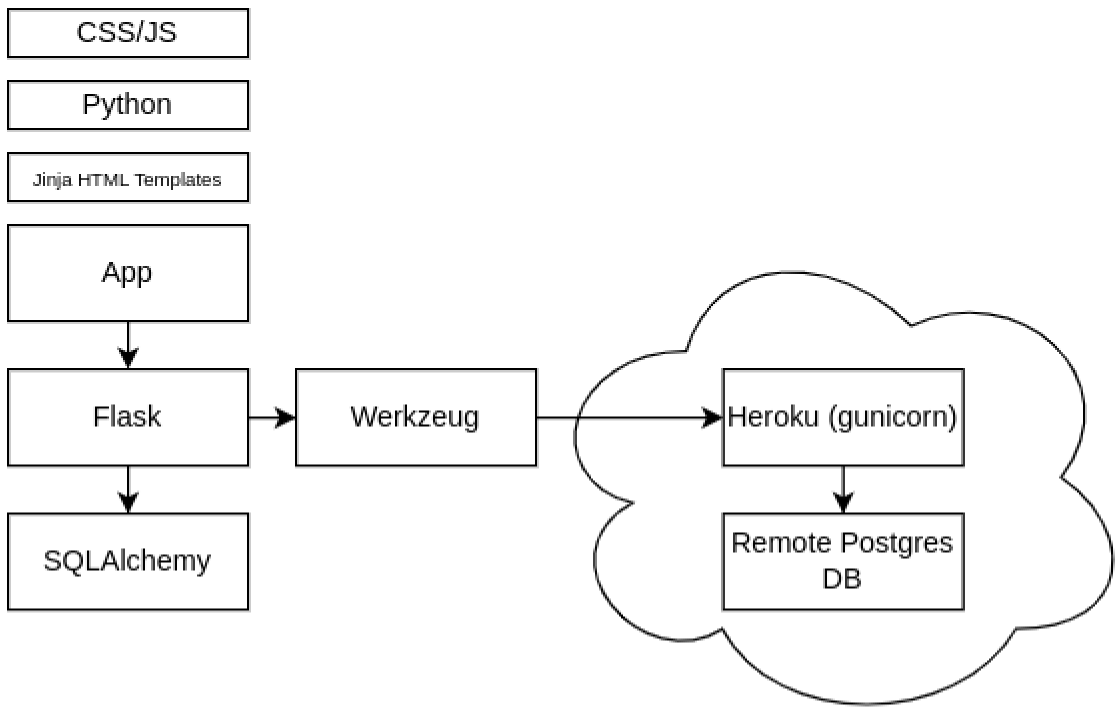


```
campsite-app::HEROKU_POSTGRESQL_RED=> \d user
                                      Table "public.user"
   Column   |          Type          | Collation | Nullable |              Default
------------+------------------------+-----------+----------+----------------------------------
 id         | integer                |           | not null | nextval('user_id_seq'::regclass)
 email      | character varying(150) |           |          |
 password   | character varying(150) |           |          |
 first_name | character varying(150) |           |          |
 hobbies    | character varying(150) |           |          |
Indexes:
    "user_pkey" PRIMARY KEY, btree (id)
    "user_email_key" UNIQUE CONSTRAINT, btree (email)
```

However, this is likely to change as work on backend functions continue.

System Implementation and Outstanding Issues

The overall app architecture is depicted below. Each component is discussed in detail in this section.



Boilerplate App Code

The boilerplate/initial code for this application was taken from the git repo techwithtim/Flask-W@views.route("/", methods=["GET", "POST"])eb-App-Tutorial. My implementation for this application heavily extends/modifies this code.

Selecting the host

Initially, AWS was considered but Heroku was chosen instead for its good documentation, ease of use and its large store of database-related add-ons. They also offer a clean CLI and web GUIs for managing applications, and they also offer some other nice features: direct integration of deployments with Github, CI pipelines (which were not designed or built out for this project), version control and easy rollbacks, and logging. The drawback of this choice is that I lose more control over configurations, and I'll spend some money on the project (though a very small amount). There are lots of options to handle scaling with Heroku, and in this sense is comparable with AWS. Although I am more likely to pay more in the long run using Heroku, for a small project the difference is negligible, and instead of spending more time trying to configure and manage the web hosting aspect, I can spend more on the development of the application.

Costs incurred so far:

- Pay for the Heroku "dyno" instance running the web application.
- Pay for the Heroku hosted database.

Selecting the RDBMS

For development, sqlite3 was chosen to get the app quickly off the ground and begin building out and debugging issues with the application. However, it was discovered that I'd need a different approach. sqlite3 databases are stored in memory, which is already a problem for a production database. The problem is made worse by the fact that Heroku dynos (compute instances) restart multiple times per day on their own, erasing any databases in the process.

I went with postgresql as the replacement as it is persistent and has some good integration options with Heroku (the chosen application host). Switching to this database was quite easy, as SQLAlchemy already supports postgres and integrating my application with Heroku's postgres service was very clean and easy. Not to mention, the documentation is quite solid.

While Heroku offers browsing of the postgresql database through their CLI, I was hoping to use a GUI similar to PHPMyAdmin or MySQLWorkbench, since I am not very familiar with interacting with SQL in the CLI.

While Heroki does not support this natively, there is an extension called Heroku Data Explorer that allows this kind of functionality. If security were a major concern for this project, I would likely find alternatives entirely to this, as the extension requires read and write privileges to the database.

Choosing the Backend and Frontend

Backend

I already have experience writing a Python backend using Django, and I wanted to build on that experience but try using Flask this time. Flask and Django are both popular web application frameworks for Python. Flask is a little more lightweight than Django, which is perfect for my needs with this application as I'm not doing anything particularly complicated, and the app itself is quite lightweight.

Flask interfaces great with other Python frameworks and libraries, such as SQLAlchemy and Werkzeug. It is a complete web applications framework, and getting going with it is extraordinarily easy.

Flask also has lots of options to handle common tasks like handling authentication (password obfuscation/hashing), error handling, etc. in the form of decorators.

For dynamic rendering of HTML, I'm using Jinja. It offers Python style programming embedded in markup files for dynamic rendering of pages. It also allows very easy extending of HTML, making it very easy to establish a common look/feel for your website by implementing a base HTML file.

Frontend

For the frontend, I chose to use CSS and Javascript. Some Bootstrap is used, which is a CSS library with extensible classes that makes quickly implementing high-quality front ends possible, but this project also contains a fair amount of custom CSS.

Map Framework

I chose to use leaflet.js to display map and markers. OpenStreetView provides the map layer displayed. Submitting a campsite using the form stores the latitude and longitude in the database. Each time the page is loaded, the application sends the client an array of all of the lat/lons for each campsite in the database to be displayed on the mark as markers (which is passed to the client as JSON arrays, which ultimately is what gets used to render the markers using Javascript). At the time of writing, I'm considering alternatives to sending this information with every home page reload as that seems incredibly expensive, especially as the database grows. I have not come up with a solution yet but my first instinct is to use cookies.