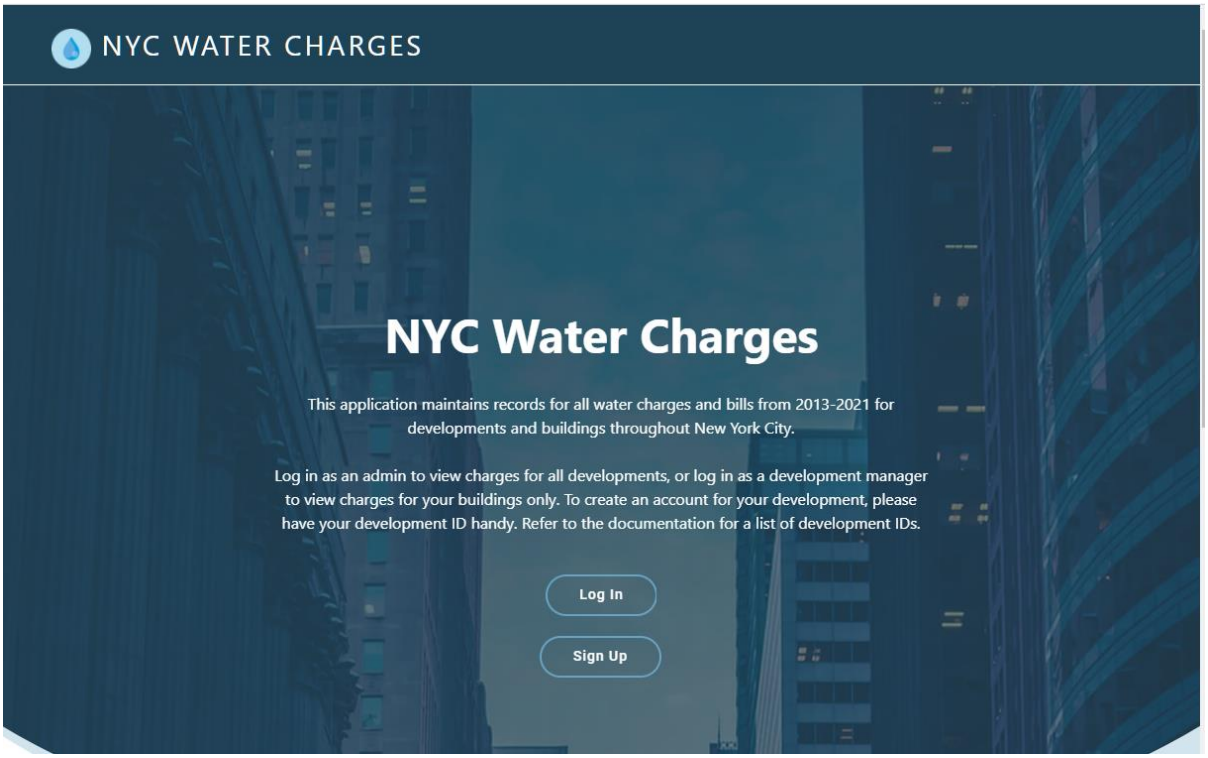
NYC Water Charges

Technology Documentation

Table of Contents

Click here for Table of Contents



About this Project

This web application was created for CISC 3810 Database Systems. The goal of this project was to create a full stack application, complete with a database layer, business layer and a user-friendly front-end interface. The data is stored in a Postgres database, hosted on Heroku. Python is used for the business layer, with Flask for the web framework, along with SQLAlchemy as an Object Relational Mapper and Flask-Login for user authentication and session management. The front-end is created in HTML 5, CSS, and JavaScript. Bootstrap is used to style the elements and to create a responsive web application. In addition, Bootstrap DataTables are used to display the rows of data. The web application is hosted on Heroku and can be accessed here.

What purpose does this software serve?

The primary purpose of this software is for the New York City Housing Authority to maintain a record of all water charges and related bills for all developments and buildings in New York City.

The secondary purpose of this software is for managers of those developments and buildings to be able to view the water service charges associated with their developments and buildings.

The software provides different levels of access for NYCHA users and for development/building users. NYCHA users have the ability to edit and delete charges. Development/Building users have the ability to view (but not edit or delete) charges.

Click here for Table of Contents

CRUD Implementation

This software incorporates all the CRUD functionality associated with a typical database application. Below is how these functionalities were implemented.

Create: This is implemented by allowing the creation of new accounts. A new user is added to the Users database table. The process is initiated when a user signs up to create a new account.

Read: This is implemented by querying the database to display all relevant information. The data is displayed and formatted using Bootstrap DataTables and jQuery.

Update: This is implemented by allowing NYCHA users to edit the charges associated with a specific bill. This process is triggered when a user with an admin account clicks on the "Edit" icon next to a bill. This feature is only available to admin users and will not appear when logged in as a development/building user.

Delete: This is implemented by allowing NYCHA users to delete a bill. This process is triggered when a user with an admin account clicks on the "Delete" icon next to the bill id. This feature is only available to admin users and will not appear when logged in as a development/building user.

Technology Stack

The technology stack is as follows:

The business layer was created in Python. I used Flask as the web framework and SQLAlchemy for Object Relational Mapping. In addition, I used Flask-Login to implement user authentication and session management.

The data was provided by NYC OpenData in the form of a csv file with 42.6K rows and 25 columns. I normalized the data and then stored it in a Postgres database, hosted on Heroku.

The front-end was created using HTML, CSS, JavaScript, Bootstrap and jQuery. In addition, Bootstrap DataTables are used to display the relevant records.

I used git and Github for version control.

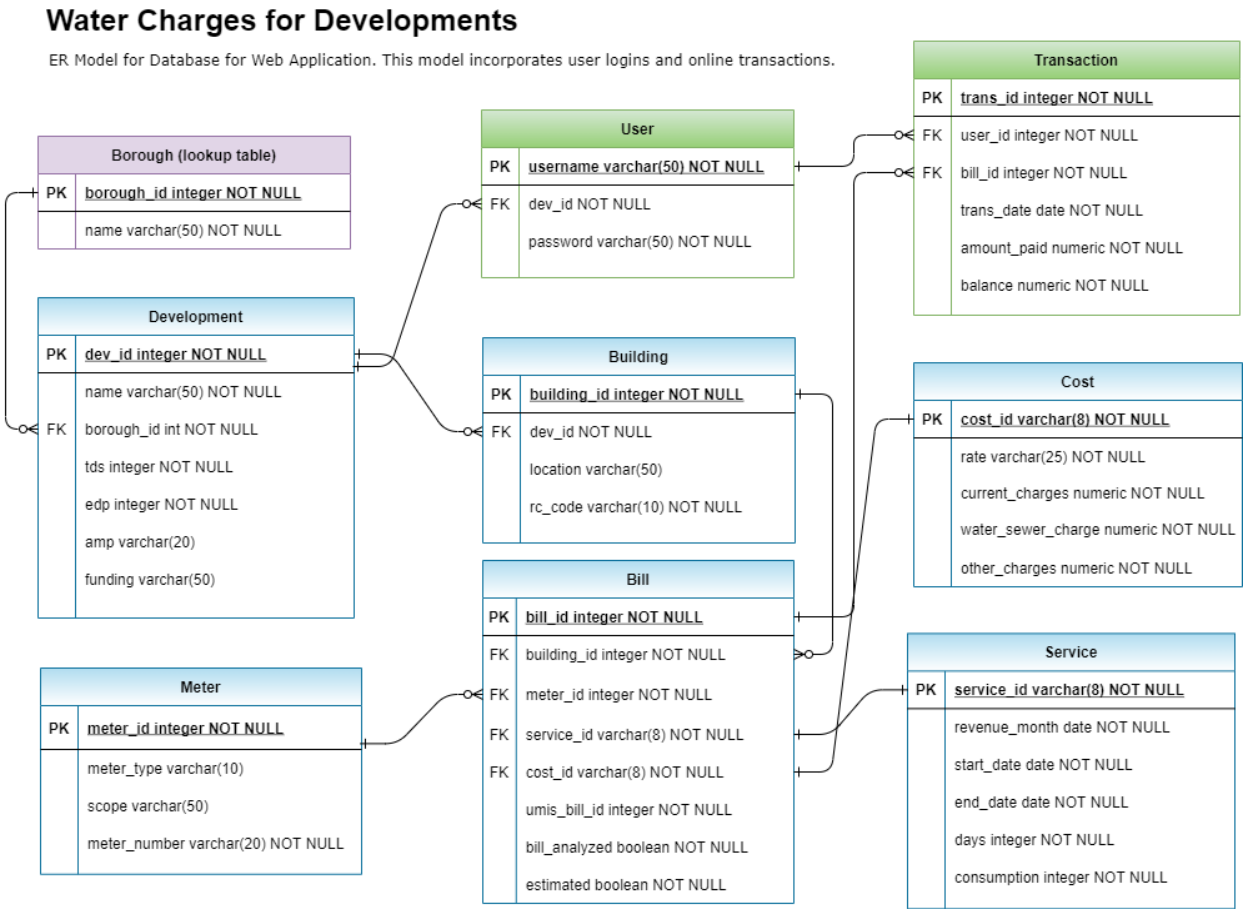
Both the database and the application are hosted on Heroku.

Click here for Table of Contents

Schemas and ER Diagram

The following tables and attributes (primary key, *foreign key*) are included in this database:

- **Development** (dev_id, name, *borough_id*, tds, edp, amp, funding)
- **Borough** (borough_id, name)
- **Building** (building_id, *dev_id*, location, rc_code)
- **Meter** (meter_id, type, scope, meter_number)
- **Service** (service_id, *vendor_id*, revenue_month, start_date, end_date, days, consumption)
- **Cost** (cost_id, *rate_id*, current_charges, water_sewer_charge, other_charges)
- **Rate** (rate_id, class)
- **Bill** (bill_id, *building_id, meter_id, service_id, cost_id*, umis_bill_id, estimated, bill_analzyed)
- **Users**(username, *dev_id*, password)
- **Transactions**(trans_id, *user_id, bill_id,* trans_date, amount_paid, balance)



Click here for Table of Contents

Extra Credit: Hosting on Heroku

Both the database and the web application are hosted on Heroku.

How to host a **web application** with a Python backend on Heroku:

1. In your project directory, run `pip freeze > requirements.txt` to gather all the Python dependencies used in the project.
2. Create a file named `Procfile` and add the following content to ensure that Heroku knows which commands it needs to run the application:
   ```
   web: gunicorn run:app
   ```
3. Create a new repository in Github to upload all your necessary files for the application.
4. Initialize your current local directory and the remote branch by running the following git commands:
   - `git init`
   - `git remote add origin address-goes-here`
5. Use the following commands to add, then commit, then push your files to your repository:
   - `git add .`
   - `git commit -m "Message goes here"`
   - `git push origin master`
6. Create a new application on Heroku.
7. Click on Deploy, and for the deployment method, choose Github.
8. Connect to Github and search for the repository created in the previous steps.
9. `Run heroku git:remote -a your-application` to add the Heroku remote to your local repository
10. Now you can deploy to Heroku by simply running the following 3 commands with the necessary parameters: `git add`, `git commit`, then `git push heroku master`.

Click here for Table of Contents

How to provision a **Postgres database** on Heroku:

1. In the Overview section of the relevant Heroku app, click on Configure Add-ons then search for Postgres.
2. Select the Heroku Postgres add-on.
3. Click on it to edit.
4. Click on Settings → View Credentials to view all the settings needed to connect to the database.
5. Log in to the database using those settings and create all the necessary tables.
6. In Python, retrieve the database connection string by calling the appropriate environment variable.
   For example: `DB_URL = os.environ['NYC_WATER_DATABASE_URL']`
7. Use that URL to configure the app in Python/Flask:
   `app.config['SQLALCHEMY_DATABASE_URI'] = DB_URL`
8. Your app can now communicate with your database.
9. Optional: use SQLAlchemy as your ORM to communicate with your database.

```
app = Flask(__name__)

db = SQLAlchemy(app)

engine = create_engine(DB_URL)
```