Internalization Instruction

For the Data Filter Web Application from Team Gabriel

Junan Zhao
UNIVERSITY OF TORONTO – CSC301 TEAM GABRIEL

INDEX

Overview

This document is a general internalization instruction document for the Toronto Transportation Traffic Data Filter Web Application (abbreviated as “the project”) created by Junan Zhao from the University of Toronto CSC301 project team Gabriel.

The project consists of two major components – the frontend website and the backend API server:

- The frontend is responsible for all visual components including the map, node and link selection interface and query parameter customization interface. The base language is ReactJS with visual components powered by material-UI and the map powered by mapbox.
- The backend is responsible for parsing frontend requests, making database function calls, parsing query results into proper formats and sending them back to the frontend. The base language is Python3 with API server powered by Flask and database functionalities powered by flask_sqlalchemy.

This document provides two internalization options – one currently used workflow with GitHub Action and Amazon Elasticbeanstalk, and one manual deployment instructions. It also has instructions on necessary modifications needed to the deployment system and the database.

Database Queries & Functions

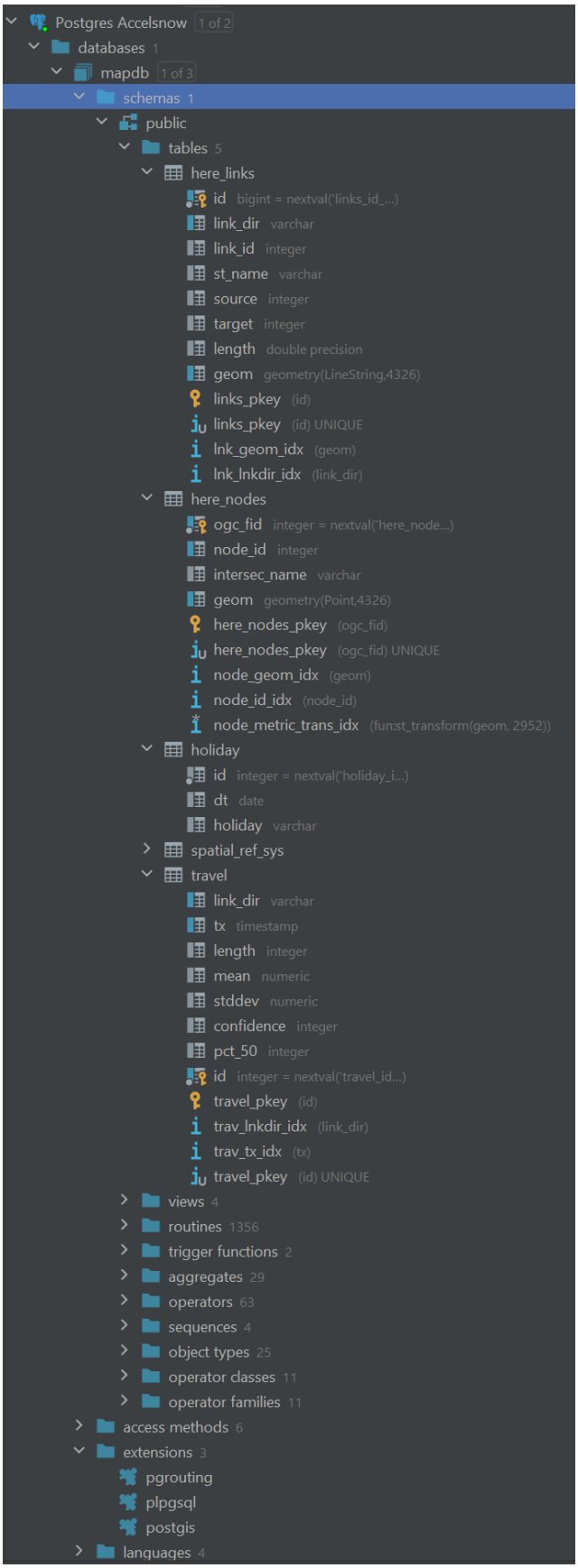
At the beginning of the development, we built a mock database based on the geojson and csv files provided. There is a database structure image on the following page illustrating our mock database structure. Should the real database have a different structure than other mock databases, system environment variables representing table names for links, nodes and travel data must be updated accordingly. Reference the System Environment Variables section of this document for more details on the required environment variables.

During the development of the project application, several functions and data types were added to the database. The queries for creating these functions and data types can be found under /backend/query_func. These functions must be present for the application to work properly.

The backend was built based on the assumption that the user assigned to it will have read-only access to the link, node, holiday, and travel data tables. After a privileged user has run the queries in query_func to create the functions, the execution permission also needs to be granted to the backend user.

Important:

- The function “fetch_trav_data_wrapper” was created because the “fetch_aggregated_travel_data” function was not optimized. The latter function executing on its own will cause the database to perform a sequential scan at the travel database, which takes a significant amount of time to execute. The wrapper function explicitly sets the “enable_deqscan” scheduler option to false, which prevents the scheduler from using a sequential scan. This is a temporary workaround, as using debug settings during production is not recommended. This function may create a permission issue, as setting scheduler options is not a read-only operation. (This problem is probably caused by calling unnest on the input argument “segments” when setting “selected_routes”)
- For rows with null intersec_name in the node table and rows with null st_name in the link table, their corresponding intersec_name and st_name will be set to “nameless road”. For any value in the travel data table with a null value, it is replaced by “no data”.


Postgres Accelsnow 1 of 2
databases 1
mapdb 1 of 3
schemas 1
public
tables 5
here_links
id bigint = nextval('links_id_...')
link_dir varchar
link_id integer
st_name varchar
source integer
target integer
length double precision
geom geometry(LineString,4326)
links_pkey (id)
links_pkey (id) UNIQUE
lnk_geom_idx (geom)
lnk_lnkdir_idx (link_dir)
here_nodes
ogc_fid integer = nextval('here_node...')
node_id integer
intersec_name varchar
geom geometry(Point,4326)
here_nodes_pkey (ogc_fid)
here_nodes_pkey (ogc_fid) UNIQUE
node_geom_idx (geom)
node_id_idx (node_id)
node_metric_trans_idx (fun:st_transform(geom, 2952))
holiday
id integer = nextval('holiday_i...')
dt date
holiday varchar
spatial_ref_sys
travel
link_dir varchar
tx timestamp
length integer
mean numeric
stddev numeric
confidence integer
pct_50 integer
id integer = nextval('travel_id...')
travel_pkey (id)
trav_lnkdir_idx (link_dir)
trav_tx_idx (tx)
travel_pkey (id) UNIQUE
views 4
routines 1356
trigger functions 2
aggregates 29
operators 63
sequences 4
object types 25
operator classes 11
operator families 11
access methods 6
extensions 3
pgrouting
plpgsql
postgis
languages 4

Deployment - Elasticbeanstalk & GitHub Action

Since the City of Toronto uses Amazon Private Cloud, it is possible to use the Amazon Elasticbeanstalk service with GitHub Action workflow to deploy the application. Two Elasticbeanstalk applications, two Amazon S3 storage buckets and a GitHub repository containing the source code are needed for this deployment method.

Create Elasticbeanstalk application and S3 Buckets on Amazon AWS

1. Decide an Amazon region that the application is going to be deployed in. The project currently uses Canada (Central) ca-central-1, which is the only region in Canada. All applications, environments, S3 buckets should be created in the same region.
2. Create an Elasticbeanstalk application for the frontend. Give it a unique name.
3. Create a new environment for the application. Select “Web server environment”.
4. Give the environment a unique name. For the platform, select “Managed platform”. Select Platform “Node.js”. The current platform branch is “Node.js 12”, and the platform version is “5.2.3”. Select “Sample application” for Application code. Click “Create environment”.
5. Create an Elasticbeanstalk application for the backend. Give it a unique name.
6. Create a new environment for the application. Select “Web server environment”.
7. Give the environment a unique name. For the platform, select “Managed platform”. Select Platform “Python”. The current platform branch is “Python 3.7”, and the platform version is “3.1.3”. Select “Sample application” for Application code. Click “Create environment”.
8. Create an Amazon S3 Bucket for the frontend. Select the same region as the applications and environments. Give it a unique name. Select block all public access.
9. Repeat step 8 and create a separate bucket with a different name for the backend.
10. After all the applications, environments and buckets are created successfully, go to the backend environment console page. Select configuration on the left and edit the Software configuration. For the proxy server, select Nginx. Change the WSGIPath to “app:app”. For the Environment properties, add all the necessary Environment properties. Reference

the System Environment Variables section of this document for more details on the required environment variables. After all the changes are made, click “Apply”.

11. Go to AWS console, click on the username at the top right corner and go to “My Security Credentials” page.
12. In the IAM console, select “Access keys (access key ID and secret access key)” from the list of folded menus. Create a new access key for this account and note the access key ID and the Secret Access Key. They are needed for security verifications later in the GitHub Action workflow. An existing access key can also be used for this purpose.

Create a GitHub repository and link the workflow to the Elasticbeanstalk applications

1. Create a GitHub repository containing the project source files.
2. IMPORTANT: Go to frontend/src/actions/, in actions.js, modify line 7-8 to define the constant variable “domain” to be the address where the backend Elasticbeanstalk environment is currently running on. This address can be found in the Elasticbeanstalk console of the backend environment. By default this address should have the format “<env name>-*.<region name>.elasticbeanstalk.com”. This is a mandatory modification; the frontend will be unable to communicate with the backend API server otherwise.
3. Go to “Settings” tab. Go to “Secrets” tab. Add a new repository secret named AWS_ACCESS that corresponds to the AWS Access Key ID. Add another repository secret named AWS_SECRET that corresponds to the AWS Secret Access Key.
4. Go to .github/workflows folder. There are 4 workflow files. The *_test.yml files are for development deployment to test servers. They can be safely removed. The *_deploy.yml files are workflow files for automated deployment to Elasticbeanstalk servers.
5. For both frontend_deploy.yml and backend_deploy.yml files, they must be edited before they are operational. For each file, under ‘env:’ line (line 2 by default), there are some variables definitions that need to be changed as follows:
    - EB_PACKAGE_S3_BUCKET_NAME: the S3 bucket name
    - EB_APPLICATION_NAME: the Elasticbeanstalk application name
    - EB_ENVIRONMENT_NAME: the Elasticbeanstalk environment name

- DEPLOY_PACKAGE_NAME: Filename of the deploy package. It must be unique each time the workflow is executed. It can be left unchanged.
- AWS_REGION_NAME: Name of the region the AWS services are deployed in

The names specified above correspond to whether it is the frontend workflow file or the backend workflow file. Excluding the DEPLOY_PACKAGE_NAME, there should be in total 6 different names for the frontend's and backend's application, environment and bucket.

6. Continue editing both files, change the branch names specified in "on: push: branches:" in both .yml files from "deploy" to the branch name to the source files are currently in.
7. Commit and push the .yml file changes. Go to "Actions" tab and wait until the triggered GitHub Action workflows finish executing. Wait 30 more seconds for the Elasticbeanstalk applications to complete their deployments.
8. Go to the AWS Elasticbeanstalk console of the frontend environment. Open in another tab the backend environment console. The health status displayed is an indicator of whether the application is running normally. If the health indicator is showing a warning or severe status, it usually implies that there is a server error, or the deployment was not successful.
9. The project application is accessible via the link displayed on the frontend console. By default this URL should have format "<env name>-*.<region name>.elasticbeanstalk.com".

Notes:

- If the backend starts sending the frontend 502 error, it usually implies that the backend failed to start. To debug on Elasticbeanstalk, go to the environment console and select Logs on the left sidebar menu. The last 100 lines of the log can be downloaded, which shall explain the error on the backend.

Deployment - Manual

This project can be built and deployed manually to a server via SSH. The server should have Python3.7+ (with python-venv and pip3) and NodeJS (with npm) installed. There are many methods for deploying the backend Flask API server and the frontend React web application. The method listed below is an example of the deployment workflow.

General Setup

1. Establish an SSH connection to the deployment server.
2. Set the required system environment variables. According to the OS running on the server, the method may vary. Reference your OS's guide for setting the variables. Reference the System Environment Variables section of this document for more details on the required environment variables.
3. Place the project files in the destinated deployment folder (via git, scp, etc.).
4. CD into the project root directory.

Backend build & deploy

1. From the project root directory, CD into folder "backend".
2. Execute command "python3 -m venv venv/" to create a python virtual environment for the backend.
3. Execute command "source venv/bin/activate" to use the virtual environment as the python interpreter for the backend.
4. Execute command "pip3 install -r requirements.txt" to install the project dependencies listed in requirements.txt.
5. Execute command "pip3 install gunicorn". Gunicorn is the service to be used to deploy a production version of the API server.
6. Execute command "gunicorn -b localhost:<port#> -w <multi-thread lim> app:app", where <port#> should be replaced by the port assigned to the API server, and <multi-thread lim> should be replaced by the maximum number of server handlers running

concurrently. This command is a blocking command; therefore, it may be helpful to consider using tools like nohup and supervisor to make it running silently at the backend.

Frontend build & deploy

1. From the project root directory, CD into folder “frontend”
2. IMPORTANT: Go to src/actions/, in actions.js, modify line 7-8 to define the constant variable “domain” to be the localhost address of the backend with the port number. This is a mandatory modification; the frontend will be unable to communicate with the backend API server otherwise.
3. Execute command “npm install” to install project dependencies for the frontend listed in package.json.
4. Execute command “npm run build” to create an optimized production build of the frontend web application.
5. Execute command “npm install pm2@latest -g” to install pm2 as the example deployment tool. This command may require root privilege.
6. Execute command “pm2 serve build <port#> -spa” to deploy the production build, where <port#> is the port number assigned to the frontend application. This will also be the port to access the project application with.
7. The project application is accessible locally on the server via “http://localhost:<frontend port#>” or remotely via “http://<server remote addr>:<frontend port#>” (if inbound traffic on the frontend port number is allowed by the firewall).

Notes:

- For security reasons, consider using reverse proxies like Nginx to avoid exposing the frontend application port by redirecting HTTP/HTTPS traffic. When redirecting traffic, it is mandatory to redirect the static build files located under frontend/build/static as well.

System Environment Variables

The backend application would check if all the necessary system environment variables have been set properly. If any of the required environment variables is missing, the backend will fail to start. The .env file at backend/ is an example of how the system environment variable should be set (the settings in .env will not work in the City of Toronto's private cloud).

Variable Name	Description
SECRET_KEY	The secret key for the flask application. Set it to a randomly generated string for security.
DATABASE_URL	The postgres database URL with pg8000 as middleware. The URL should be in the following format: postgresql+pg8000://<username>:<password>@ <db_addr>:<db_port>/<db_name> Example: postgresql+pg8000://test:password@xxx.com:5432/mapdb
LINK_TABLE_NAME	The table name for the link data.
NODE_TABLE_NAME	The table name for the node data.
TRAVEL_DATA_TABLE_NAME	The table name for the travel data.
POSTGIS_GEOM_SRID	The geom SRID used by default by the database.
TEMP_FILE_LOCATION	The directory to place temporary files generated during file construction. If the path string starts with '/', it will be an absolute path; otherwise, it will be relative to the backend's current working directory.
KEEP_TEMP_FILE	Set to 'true' to keep the temporary files generated. Set to 'false' to delete temporary files after sending them.

DB_DATA_START_DATE	The date of the first entry in the database. This date string should be in the format “%Y-%m-%d %H:%M” (example: “2017-01-01 00:00”. The format is the same as FULL_DATE_TIME_FORMAT in backend/__init__.py). Used to bound user input.

Miscellaneous

In backend/__init__.py:

- The fields for the query result from the database function “fetch_aggregated_travel_data” are defined in DB_TRAVEL_DATA_QUERY_RESULT_FORMAT variable. It is a dictionary mapping the names of the columns to their variable type and indexes (column index of the query result, -1 means it is not part of the query result but a generated column in the data file).
- The database ORM models for the link table, node table and travel data table can be found in backend/app/models.py. They are powered by SQLAlchemy.

- Make sure in backend/requirements.txt that the version of pg8000 is less than 1.16.6 (exclusive). There are compatibility issues between pg8000=1.16.6 and the latest SQLAlchemy library.
- The backend process should have read/write permission to the path defined in the system environment variable TEMP_FILE_LOCATION.
- The backend can also run as a standalone API server. Reference doc/backend-OpenAPIdoc.html for the API specifications.
- The lower bound of user input date is fetched from the system environment variable DB_DATA_START_DATE, while the upper bound is calculated upon request. If the current time is before 17:30, the upper bound is 3 days ago at 23:59; if the current time is 17:30 or after, the upper bound is 2 days ago at 23:59.