Scraping and Ingest - User Manual

Gabriel Crisnejo

February 23, 2025

1 Introduction

This document provides a comprehensive overview of the architecture and functionality of the web scraping project. It outlines the process of collecting news articles from an online source, processing the data, storing it in BigQuery for structured querying, and deploying the entire pipeline to Google Cloud Run. The document covers each phase of the project, from data extraction using Selenium to the final deployment steps on Google Cloud, ensuring a seamless integration for continuous operation in a cloud environment.

2 General considerations

Ensure your Google Cloud account has the necessary permissions to access Cloud Run, BigQuery and Artifact Registry. Create a project on Google Cloud Platform and a Docker repository in Google Artifact Registry called my-docker-repo.

3 Project Structure

The project is organized into the following structure, where each file and directory serves a specific purpose in the web scraping, data processing, and deployment pipeline:

```
.
|-- deploy.sh               # Deployment script for Google Cloud Run
|-- Dockerfile              # Dockerfile for containerizing the application
|-- main.py                 # Entry point for running the application
|-- README.md               # Project documentation and setup instructions
|-- requirements.txt        # Python dependencies for the project
|-- .env                        # Environment variables setting
|-- docs                        # Documentation
|   |-- user_manual.pdf             # User manual
|-- src                         # Source code directory
|   |-- bigquery.py             # Module for uploading data to BigQuery
|   |-- post_processing.py      # Module for post-processing scraped data
|   |-- scraping.py             # Module for scraping data from the target website
|   |-- settings.py             # Configuration settings for the project
|-- webscrapingproject-451423-bfb0a6dca447.json
                                # Google Cloud service account credentials
```

4 Installation and Setup

To run the project locally, follow these steps:

```
# Clone the repository
$ git clone https://github.com/GabrielCrisnejo/
  scraping_ingest.git
$ cd scraping_ingest

# Create a virtual environment and activate it (optional)
$ python -m venv venv
$ source venv/bin/activate

# Install the required dependencies
$ pip install -r requirements.txt

# Run the project locally
$ python main.py
```

5 Deploying on Google Cloud Run

To deploy the project on Google Cloud Run, follow these steps:

```
# Make sure you have the deploy script executable
$ chmod +x deploy.sh

# Deploy the application to Cloud Run
$ ./deploy.sh
```

6 Interacting with the Service

After the deployment is successful, the service will be accessible via the Cloud Run URL. You can interact with the service using the following steps:

Check the service status:

To check if the service is up and running, send a GET request to the root endpoint (`/`):

```
$ curl https://<your-cloud-run-url>/
```

For example,

```
$ curl https://web-scraper-challenge-109720451514.us-central1
    .run.app
```

This will return the status of the service, confirming if it's running or encountering any issues.

Trigger the service:

To trigger the main functionality of the application, send a GET request to the `/run` endpoint:

```
$ curl https://<your-cloud-run-url>/run
```

For example,

```
$ curl https://web-scraper-challenge-109720451514.us-central1
    .run.app/run
```

This will initiate the scraping and processing tasks, and the response will confirm the operation has started or provide an error message.

Note

If you run the application locally, it will run on the localhost so we need to do the following to interact with the service,

```
$ curl https://http://127.0.0.1:8080/
$ curl https://http://127.0.0.1:8080/run
```

7 Conclusion

This document provides a detailed guide on the project's functionality, installation, execution, and deployment on Google Cloud Run. By following these steps, you can easily deploy the application and interact with it via HTTP requests. Additionally, ensure that all necessary permissions and configurations are set up in Google Cloud to guarantee smooth deployment and operation.

A File Descriptions and Details

This appendix provides detailed descriptions of the files and modules included in the project.

A.1 scraping.py

This module performs web scraping of news articles from Yogonet International. It uses Selenium to automate interactions with the website and extract the following information:

- Title of the article
- Kicker (text above the main headline)
- Image URL
- Link to the full article

The scraping process begins by configuring a headless Chrome browser using `Selenium` and `webdriver-manager` to manage the Chrome driver.

This data is stored in an organized structure (a Pandas DataFrame) and saved to a CSV file at the location specified by `SCRAPED_DATA_PATH`.

The code handles potential errors during the data extraction process using exception handling and ensures that the browser is properly closed at the end of the process, even if an error occurs.

This approach allows for efficient and reliable scraping of news articles from the specified website.

A.2 post_processing.py

This module processes the scraped data and performs the following tasks:

- Word and character count for both title and kicker.
- Identification of capitalized words in the title.
- Validation of article links and image URLs.
- Handling missing images and missing data.

The process begins by reading the scraped data from the CSV file specified by `SCRAPED_DATA_PATH`. The module then applies several transformations and calculations to the data:

- **Word count (title):** The total number of words in the title is calculated using the `split()` method.
- **Character count (title):** The total number of characters in the title is calculated using the `len()` function.
- **Capitalized words (title):** A list of words in the title that start with a capital letter is generated using `istitle()`.
- **Word count (kicker):** The number of words in the kicker is calculated similarly to the title, with special handling for missing data.
- **Character count (kicker):** The character count for the kicker is determined, with a check for missing data.
- **Image URL validity:** The validity of the image URL is checked using the `is_valid_url()` function, which parses the URL and ensures it has a valid scheme and netloc.
- **Link URL validity:** Similarly, the validity of the article link URL is checked.
- **Missing images:** The presence of missing images is checked by identifying empty or NaN image entries.

After performing these transformations, the processed data is saved to a new CSV file at the location specified by `POST_PROCESSED_DATA_PATH`.

A.3 bigquery.py

This module uploads the processed data to Google BigQuery. It performs the following tasks:

- Authenticates using a service account for accessing Google Cloud resources.
- Checks for the existence of a dataset and creates it if necessary.
- Defines the schema for the BigQuery table and creates the table if it doesn't exist.
- Loads the processed data from a CSV file into the BigQuery table.

The process begins by constructing the fully qualified name of the BigQuery table using the project ID, dataset ID, and table ID. It then authenticates the client using a service account file (`SERVICEACCOUNT.json`) and establishes a connection to the BigQuery project.

The script checks if the dataset exists in the specified project. If it does, a confirmation message is printed. If the dataset does not exist, the script creates it, specifying the location using the `REGION`.

Next, the schema for the BigQuery table is defined, specifying the field names and types. If the table already exists, a message is printed. If the table does not exist, the script creates it using the defined schema.

Once the dataset and table are confirmed or created, the processed data from the CSV file (located at `POST_PROCESSED_DATA_PATH`) is loaded into the BigQuery table. The `WRITE_TRUNCATE` disposition is used to overwrite the table with the new data.

The data is uploaded via the BigQuery client, and the upload job is monitored using `job.result()` to ensure that the operation completes successfully.

A.4 settings.py and .env

The `settings.py` file is used to load configuration variables from the environment, allowing the application to be easily configured for different environments. The file utilizes the `dotenv` library to load values from a `.env` file, which contains sensitive or environment-specific settings. The configuration includes the following parameters:

- `URL:` The URL of the website to scrape.
- `SCRAPED_DATA_PATH:` The path where the scraped data will be stored, defaulting to `/tmp/scraped_data.csv`.
- `POST_PROCESSED_DATA_PATH:` The path where the post-processed data will be saved, defaulting to `/tmp/post_processed_data.csv`.
- `PROJECT_ID:` The Google Cloud project ID, defaulting to `webscrapingproject-451423`.
- `DATASET_ID:` The BigQuery dataset ID where the data will be uploaded, defaulting to `dataset_webscraping`.
- `TABLE_ID:` The BigQuery table ID for the scraped data, defaulting to `table_webscraping`.

- `SERVICEACCOUNT`: The service account used for authenticating with Google Cloud, defaulting to `webscrapingproject-451423-bfb0a6dca447`.
- `REGION`: The Google Cloud region where the resources will be deployed, defaulting to `US`.

The values for these variables are read from the `.env` file using `os.getenv()`. This allows the application to be easily configured without modifying the code. For example, the following environment variables are defined in the `.env` file:

```
URL=https://www.yogonet.com/international/
SCRAPED_DATA_PATH=/tmp/scraped_data.csv
POST_PROCESSED_DATA_PATH=/tmp/post_processed_data.csv

PROJECT_ID=webscrapingproject-451423
SERVICEACCOUNT=webscrapingproject-451423-bfb0a6dca447

REGION=us-central1
SERVICE_NAME=web-scraper-challenge-7
REPO_NAME=my-docker-repo
IMAGE_NAME=us-central1-docker.pkg.dev/$PROJECT_ID/$REPO_NAME/web-scraper
```

These environment variables are automatically loaded when the application starts, ensuring that the necessary configurations are set without hard-coding values into the codebase. This approach is essential for maintaining flexibility and security, especially when working with sensitive information like service accounts or project IDs.

A.5 requirements.txt

Lists all required Python dependencies and their versions.

A.6 main.py

This module orchestrates the execution of the entire data pipeline using Flask. It provides the following functionalities:

- A health check endpoint at `/` to verify that the container is running.
- An endpoint at `/run` that, when accessed via a GET request, triggers the full pipeline, executing:

  - Web scraping using `scrape_data()`.
  - Data processing using `process_data()`.
  - Uploading of processed data to BigQuery using `upload_to_bigquery()`.

- Runs a background thread that keeps the Flask server active while preventing automatic restarts.

A.7 Dockerfile

The `Dockerfile` describes the process of setting up a Docker container for the application. The following steps are performed:

- **Base Image:** The container uses the `python:3.12.2` image as the base, which includes Python 3.12.2.
- **Install Dependencies:** The necessary system dependencies for running the application, including libraries for graphical operations (such as `libgtk-3-0` and `libnss3`) are installed using `apt-get`. These libraries are required for running Selenium and Chrome in the container.
- **Install Google Chrome:** Google Chrome is installed by downloading the `.deb` package and using `dpkg` to install it. `apt-get` is then used to resolve any missing dependencies.
- **Set Working Directory:** The working directory is set to `/app`, where the application code will reside.
- **Copy Application Code:** The application files are copied into the container's working directory.
- **Install Python Dependencies:** The Python dependencies listed in `requirements.txt` are installed using `pip` with the `--no-cache-dir` option to avoid caching.
- **Expose Port:** The container exposes port 8080, which can be used by the application to communicate with external services.
- **Run Application:** The default command is set to `python main.py`, which runs the main Python application when the container starts.

This `Dockerfile` sets up an environment with Python and Chrome for running a web scraping application using Selenium.

A.8 deploy.sh

The `deploy.sh` script automates the process of building and deploying the Docker container to Google Cloud Run. It performs the following steps:

- **Set -e:** The script begins with `set -e`, which ensures that the script stops execution if any command fails.
- **Load Environment Variables:** The `source .env` command loads environment variables from the `.env` file to make them available to the script.
- **Build Docker Image:** The script uses `docker build` to build a Docker image from the `Dockerfile` in the current directory. The image is tagged with the name defined by the `IMAGE_NAME` variable.
- **Authenticate with Google Artifact Registry:** The script authenticates with Google Artifact Registry by running `gcloud auth configure-docker`, which allows Docker to interact with Google's container registry.
- **Push Docker Image:** The built image is pushed to the Google Artifact Registry using `docker push` with the image name defined by `IMAGE_NAME`.
- **Deploy to Cloud Run:** The script deploys the Docker container to Google Cloud Run using `gcloud run deploy`. The deployment includes several configurations, such as:
  - `--image:` Specifies the Docker image to deploy.
  - `--platform managed:` Uses Google Cloud Run's fully managed platform.
  - `--region:` Specifies the Google Cloud region to deploy the service.
  - `--allow-unauthenticated:` Allows unauthenticated access to the service.
  - `--port 8080:` Exposes port 8080 for the container.
  - `--memory 2Gi:` Allocates 2 GB of memory for the service.
  - `--set-env-vars:` Sets environment variables that will be available to the deployed application, such as `SCRAPED_DATA_PATH`, `POST_PROCESSED_DATA_PATH`, and `URL`.

- **Completion Message:** Once the deployment is finished, the script prints a success message: `Deployment completed successfully!`.

This script streamlines the process of building, authenticating, and deploying the application to Google Cloud Run, allowing for a simple and repeatable deployment process.