Weather Data Collection Module

`app.py` (To trigger the Weather Data Mining process)

```
from dotenv import load_dotenv
from weather_data_miner import WeatherDataMiner

# Load environment variables from .env file
load_dotenv()

def run():
    WeatherDataMiner().run()

if __name__ == "__main__":
    run()
```

`weather_data_miner.py` (Main code that does the collection of weather data)

```
import requests
import csv
from datetime import datetime, timedelta
import os
import json
import logging
import time
from constants.constants import HOURLY, MODES, OUTPUT_DATA_FOLDER, FORECAST

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -
%(message)s')

class WeatherDataMiner:
    """A class to miner weather data from the Weatherbit API and save it to CSV
files."""

    def __init__(self):
        """
```

```
        Initializes the WeatherDataMiner with environment variables.
        """
        self.base_url = os.getenv('BASE_URL')
        self.api_keys = os.getenv('API_KEYS').split(',')
        self.mode = os.getenv('MODE', HOURLY).lower()
        if self.mode not in MODES.keys():
            raise Exception(f"Invalid mode selected. Select a valid mode.
({MODES.keys()})")
        self.current_key_index = 0
        self.oldest_date = datetime.strptime(os.getenv('OLDEST_DATE'),
'%Y-%m-%d')
        self.days_per_request = int(os.getenv('DAYS_PER_REQUEST', 28))
        self.save_checkpoint_months = int(os.getenv('SAVE_CHECKPOINT_MONTHS',
6))
        self.max_retries = int(os.getenv('MAX_RETRIES', 3))
        self.retry_delay_seconds = int(os.getenv('RETRY_DELAY_SECONDS', 5))
        self.locations = json.loads(os.getenv('LOCATIONS'))
        self.api_key = self.get_api_key()
        self.date_tracking = self.load_date_tracking()
        self.forecast_hours = int(os.getenv('FORECAST_HOURS', 240))

    def run(self):
        """Runs the data mining process for all configured locations."""
        for location in self.locations:
            try:
                self.mine_location(location)
                logging.info(f'All data collection completed for {location}.')
            except Exception as e:
                logging.error(f'An error occurred while processing location
{location["name"]}: {e}')
                break

    def get_api_key(self):
        """Retrieve the next API key from the list."""
        if self.current_key_index < len(self.api_keys):
            return self.api_keys[self.current_key_index]
        return None

    def mine_location(self, location):
        """Mines historic weather data for a single location and saves it to a
CSV file."""
        lat = location['lat']
        lon = location['lon']
        location_name = location['name']
        filename = f'{location_name}_weather_data_{self.mode}.csv'

        start_date_str = self.date_tracking.get(location_name,
self.oldest_date.strftime('%Y-%m-%d'))
        start_date = datetime.strptime(start_date_str, '%Y-%m-%d')
```

```
        end_date = datetime.utcnow()
        all_data = []

        while start_date < end_date:
            current_end_date = min(start_date +
timedelta(days=self.days_per_request), end_date)
            data = self.get_weather_data_with_retry(lat, lon,
start_date.strftime('%Y-%m-%d'), current_end_date.strftime('%Y-%m-%d'))
            if data is None:
                if len(all_data) == 0:
                    logging.warning(f'Failed to fetch data after
{self.max_retries} retries. No data collected for {location}')
                    raise
                logging.warning(f'Failed to fetch data after {self.max_retries}
retries. Saving collected data and terminating.')
                self.save_to_csv(all_data, filename)
                if self.mode != FORECAST:
                    self.date_tracking[location_name] =
start_date.strftime('%Y-%m-%d')
                    self.save_date_tracking(self.date_tracking)
                break
            all_data.extend(data)
            if (current_end_date - self.oldest_date).days >=
self.save_checkpoint_months*30 or current_end_date == end_date:
                self.save_to_csv(all_data, filename)
                all_data = []
                if self.mode != FORECAST:
                    self.date_tracking[location_name] =
current_end_date.strftime('%Y-%m-%d')
                    self.save_date_tracking(self.date_tracking)
            start_date = current_end_date
        if len(all_data):
            logging.info(f'Data saved to {filename} from {start_date} to
{end_date}')

    def get_weather_data_with_retry(self, lat, lon, start_date, end_date):
        """Makes an API call to retrieve weather data with rate limiting and
retries."""
        attempt = 0
        while attempt < self.max_retries:
            self.api_key = self.get_api_key()
            if not self.api_key:
                logging.error('All API keys have been rate limited. Stopping
application.')
                raise Exception('All API keys rate limited')  # Or use a more
specific exception
            try:
                if self.mode == FORECAST:
                    url =
```

```
f'{self.base_url}{MODES[self.mode]["URL"]}?tz=local&lat={lat}&lon={lon}&key={sel
f.api_key}&hours={self.forecast_hours}'
                else:
                    url =
f'{self.base_url}{MODES[self.mode]["URL"]}?tz=local&lat={lat}&lon={lon}&start_da
te={start_date}&end_date={end_date}&key={self.api_key}'
                response = requests.get(url)
                response.raise_for_status()  # This will raise an exception for
HTTP errors
                return response.json()['data']
            except requests.exceptions.HTTPError as e:
                if e.response.status_code == 429:
                    logging.warning(f'Rate limit exceeded for API key
{self.api_key}. (attempt {attempt + 1})')
                    # Move to the next API key
                    if self.current_key_index >= len(self.api_keys):
                        # If all keys are exhausted, stop the application
                        logging.error('All API keys have been rate limited.
Stopping application.')
                        raise
                    if attempt < self.max_retries - 1:
                        time.sleep(self.retry_delay_seconds)
                        attempt += 1
                    else:
                        self.current_key_index += 1
                        attempt = 0
                else:
                    logging.warning(f'HTTPError for URL {url}: {e}')
                    raise
            except requests.exceptions.RequestException as e:
                if attempt < self.max_retries - 1:
                    time.sleep(self.retry_delay_seconds)
                else:
                    logging.error(f'RequestException for URL {url}: {e}')
                    raise

    def save_to_csv(self, data, filename):
        """Saves the data to a CSV file, flattening nested objects."""
        if not data:
            return
        filepath = f'{OUTPUT_DATA_FOLDER}/{filename}'
        # Prepare the CSV file for writing
        with open(filepath, mode='a', newline='') as file:
            # If the file is empty, write the header
            if file.tell() == 0:
                # Extract fieldnames from the first data entry
                # This includes nested fields like 'weather.icon'
                fieldnames = self.get_fieldnames(data[0])
                writer = csv.DictWriter(file, fieldnames=fieldnames)
```

```
                writer.writeheader()
            else:
                # No need to write the header
                writer = csv.DictWriter(file,
fieldnames=self.get_fieldnames(data[0]))

            # Write the data rows, flattening each entry
            for entry in data:
                flat_entry = self.flatten_data(entry)
                writer.writerow(flat_entry)

    def get_fieldnames(self, data_entry):
        """Recursively extracts field names from a nested data entry."""
        fieldnames = []
        for key, value in data_entry.items():
            # If the value is a dictionary, recurse
            if isinstance(value, dict):
                sub_fieldnames = self.get_fieldnames(value)
                # Prefix the nested keys with the current key
                fieldnames.extend([f"{key}.{sub_key}" for sub_key in
sub_fieldnames])
            else:
                fieldnames.append(key)
        return fieldnames

    def flatten_data(self, data_entry):
        """Flattens a nested data entry into a single dictionary with
dot-separated keys."""
        flat_data = {}
        for key, value in data_entry.items():
            # If the value is a dictionary, recurse
            if isinstance(value, dict):
                sub_flat_data = self.flatten_data(value)
                # Prefix the nested keys with the current key
                for sub_key, sub_value in sub_flat_data.items():
                    flat_data[f"{key}.{sub_key}"] = sub_value
            else:
                flat_data[key] = value
        return flat_data

    def load_date_tracking(self):
        """Loads the date tracking from a JSON file."""
        if os.path.exists(MODES[self.mode]['TRACKER']):
            with open(MODES[self.mode]['TRACKER'], 'r') as file:
                return json.load(file)
        else:
            return {}

    def save_date_tracking(self, date_tracking):
```

```
        """Saves the date tracking to a JSON file."""
        with open(MODES[self.mode]['TRACKER'], 'w') as file:
            json.dump(date_tracking, file)
```

`constants.py` (Holds constants value of the code)

```
OUTPUT_DATA_FOLDER = "data"
HOURLY = "hourly"
SUB_HOURLY = "sub_hourly"
FORECAST = "forecast"
TRACKER_FOLDER = "tracker"

MODES = {
  HOURLY: {
    "URL": "/history/hourly",
    "TRACKER": f"{TRACKER_FOLDER}/data_tracker_{HOURLY}.json"
  },
  SUB_HOURLY: {
    "URL": "/history/subhourly",
    "TRACKER": f"{TRACKER_FOLDER}/data_tracker_{SUB_HOURLY}.json"
  },
  FORECAST: {
    "URL": "/forecast/hourly",
    "TRACKER": f"{TRACKER_FOLDER}/data_tracker_{FORECAST}_{HOURLY}.json"
  }
}
```

`fix_weather_data.py` (Fixing the column order)

```
import pandas as pd
import os

def is_alphanumeric_with_numeric(s):
    if s.isalnum() and any(char.isdigit() for char in s) and any(char.isalpha()
for char in s):
        return True
    else:
```

```
        return False

def fix_columns(row):
    if is_alphanumeric_with_numeric(str(row['weather.description'])): #desc is
icon
        if str(row['weather.icon']).isdigit(): #icon is code then code is desc
            row['weather.icon'], row['weather.description'], row['weather.code']
= row['weather.description'], row['weather.code'], row['weather.icon']
        else:
            row['weather.icon'], row['weather.description'] =
row['weather.description'], row['weather.icon']
    elif str(row['weather.description']).isdigit():# desc is code
        if is_alphanumeric_with_numeric(str(row['weather.code'])):# code is icon
then icon is desc
            row['weather.icon'], row['weather.description'], row['weather.code']
= row['weather.code'], row['weather.icon'], row['weather.description']
        else:
            row['weather.description'], row['weather.code'] =
row['weather.code'], row['weather.description']
    return row

# Directory containing the CSV files
directory = './data/'

# Iterate over each file in the directory
for filename in os.listdir(directory):
    if filename.endswith(".csv"):
        file_path = os.path.join(directory, filename)

        # Load the CSV file
        df = pd.read_csv(file_path)

        # Apply the fix_columns function
        df = df.apply(fix_columns, axis=1)

        # Save the DataFrame back to the CSV file
        df.to_csv(file_path, index=False)

print("All CSV files have been successfully fixed.")
```

.env (Project Environment File)

```
# .env file
API_KEYS=Token1, Token2, Token3
OLDEST_DATE=2020-01-01
DAYS_PER_REQUEST=28
SAVE_CHECKPOINT_MONTHS=6
MAX_RETRIES=3
RETRY_DELAY_SECONDS=5
LOCATIONS=[{ "name": "MCO", "lat": 28.424618, "lon": -81.310753 },{ "name":
"SYR", "lat": 43.111943, "lon": -76.114139 },{ "name": "ORD", "lat": 41.978611,
"lon": -87.904724 },{ "name": "JFK", "lat": 40.641766, "lon": -73.780968 }]
BASE_URL=https://api.weatherbit.io/v2.0
MODE=hourly
FORECAST_HOURS=168
```