**How to set up and run the project locally (including Celery and broker setup).**

**For Windows Users:**

1. After cloning or downloading the repo, it is recommended that you create a virtual environment first.   
   “python -m venv venv”.
2. After setting up venv and activating it, there is a file requirements.txt which contains all the libraries with versions that I have used for this project. You can install those with   
   “pip install -r requirements.txt”.
3. After the installation of libraries, there is a file of “.env” where we keep our private settings like secret key and DB creds. I have used mysql but you may configure any other database by adding the db\_name, username, password, host, and port in .env file and settings.py. You can either use cmd or phpMyAdmin for creating DB of mysql or simply uncomment the default engine in settings.py for sqlite3 and comment the mysql settings.
4. Migrate the migrations and run the project.   
   “python manage.py migrate” and “python manage.py runserver”.
5. For broker setup, you have to install redis in your PC, which can be [downloaded](https://github.com/13530361335/windows-software/blob/master/Redis-x64-3.2.100.msi) and install easily. After installing and adding the path in environmental variables. Open cmd, type “redis-server” to turn on the redis server.
6. Celery library is already installed, also the settings but to turn celery worker on we have to open another terminal and type   
   “celery -A amazon\_scrapper worker –pool=solo -l info”,   
   although params may differ in this command like –pool or any addition in param like concurrency or --loglevel, but this worker can also do the job.
7. Turn on Django Celery beat, open another terminal and type   
   “celery -A amazon\_scrapper beat -l info --scheduler django\_celery\_beat.schedulers:DatabaseScheduler”   
   this will help us for the periodictasks.
8. For Django admin, open terminal in the project’s folder, type   
   “python manage.py createsuperuser”   
   enter details as required and access the [admin panel](http://127.0.0.1:8000/admin).
9. Now the project is up and running.

**Instructions for scheduling and managing the periodic tasks.**

Periodic tasks are managed using Django Celery Beat. You can set up periodic tasks using the following approach:

1. Use the “setup\_periodic\_tasks” function in *products/tasks.py* to create tasks that run at specified intervals (in our case it’s every 6 hours). This function is called at application startup.
2. You can manage periodic tasks through the Django admin interface under the “Periodic tasks” app.

**How the web scraping is implemented, including any anti-scraping measures.**

The web scraping functionality is implemented using the “requests” and “BeautifulSoup” libraries, allowing retrieval and parsing of Amazon product information based on brand names. The implementation is organized into three primary functions to separate concerns: *scrape\_amazon\_products, parse\_product\_list, and fetch\_product\_sku*.

**Implementation:**

**scrape\_amazon\_products(brand\_name):**

1. Constructs the search URL using the provided brand name and iterates through paginated results.
2. Sends an HTTP GET request to fetch each page, utilizing a randomly selected user-agent header to mimic different browser requests.
3. Passes the page content to *parse\_product\_list* for data extraction.
4. Adds a random delay between requests to avoid detection.
5. Continues to the next page until no products are found.

**parse\_product\_list(soup):**

1. Parses product information from a single page.
2. For each product found, retrieves its name, ASIN, and image URL.
3. Calls *fetch\_product\_sku* for each ASIN to obtain the SKU from the individual product page, appending results to the product list.

**fetch\_product\_sku(asin):**

1. Accesses the individual product page for a given ASIN.
2. If the request is successful and no CAPTCHA is encountered, it parses the page for SKU information.
3. Returns the SKU if found; otherwise, logs the failure and returns None.

**Anti-Scraping Measures**

1. The code uses user-agent strings to mimic requests from different browsers.
2. A random delay is introduced between requests to avoid making too many requests in a short time.
3. If a request fails, an exception is raised to retry or log the error.

**Any assumptions or design decisions.**

* The project assumes that the Amazon product pages will follow a consistent HTML structure; changes to this structure may require updates to the scraping logic.
* The use of Redis as a broker is based on its performance and support for Celery; other brokers can be used but may require different configurations.
* The decision to scrape product details at regular intervals is based on the assumption that product data can change frequently, necessitating updates in the database.
* The use of Django's ORM simplifies database interactions, but the implementation could be adjusted for raw SQL queries if performance becomes an issue.

**Additionally:**

* I have implemented logging to track.
* Added Search functionality for the list of products, also the pagination to prevent load time for the list of products.
* I have also used caching in scraping the products data.