Task:-

1. **Uses Node.js with Puppeteer and Chromium to scrape a user-specified URL.**
2. **Uses Python (with a lightweight web framework like Flask) to host the scraped content.**
3. **Demonstrates the combined power of Node.js for browser automation and Python for serving content, while keeping the final image lean.**

Objective:-

- **Multi-Stage Build:** Develop a Dockerfile that contains at least two stages:
    - A **build stage** (or scraper stage) based on a Node.js image that installs Chromium and Puppeteer, executes a script to scrape data from any provided URL, and saves the output (e.g., a JSON file).
    - A **final stage** based on a Python image that copies the scraped output and runs a web server to host the content.
- **Puppeteer & Chromium:** Properly install Chromium (or Google Chrome) and configure Puppeteer so that your Node.js script can run headless browser operations.
- **Dynamic Scraping:** Your scraper should accept a URL parameter (either via an environment variable or command-line argument) and then scrape the specified site.
- **Hosting:** Implement a simple web server (using Python and Flask) that reads the scraped output and displays it as JSON when accessed via a web browser.
- **Containerization:** The final Docker container should expose a port and allow users to access the scraped content over HTTP.

Requirements

1. **Node.js Scraper Stage:**
    - Base image: Use a Node.js (e.g., `node:18-slim`) image.
    - Install required dependencies (Chromium, fonts, etc.) using apt.
    - Configure Puppeteer to skip its bundled Chromium download (using `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD`) and use the installed version.
    - Create a script (`scrape.js`) that:
        - Accepts a URL (via an environment variable `SCRAPE_URL` or similar).
        - Launches Puppeteer in headless mode with the proper flags (`--no-sandbox`, etc.).
        - Navigates to the given URL.
        - Extracts data (for example, the page title and first heading).
        - Writes the scraped data to a file (e.g., `scraped_data.json`).
2. **Python Hosting Stage:**
    - Base image: Use a Python (e.g., `python:3.10-slim`) image.
    - Copy the `scraped_data.json` from the previous stage.
    - Implement a simple Flask application (`server.py`) that:
        - Reads the JSON file.

        - Provides an HTTP endpoint (e.g., at `/`) that returns the scraped content as JSON.
    - Expose port 5000 (or another port of your choice).
3. **Multi-Stage Dockerfile:**
    - Combine both stages in a single Dockerfile using multi-stage builds.
    - Ensure that the final image is as slim as possible by only including necessary runtime files.
    - The container should start the Python web server when run.
4. **Usage Documentation:**
    - Provide a README file that describes:
        - How to build the Docker image.
        - How to run the container.
        - How to pass the URL to be scraped (via environment variables or command-line arguments).
        - How to access the hosted scraped data.

Deliverables

- **Dockerfile:** A multi-stage Dockerfile that builds the complete application.
- **scrape.js:** Node.js script that uses Puppeteer to scrape a given URL.
- **server.py:** Python Flask application that serves the scraped data.
- **package.json:** For Node dependencies.
- **requirements.txt:** For Python dependencies.
- **README.md:** Documentation with build and run instructions, including how to supply the URL.

Evaluation Criteria

- **Correctness:** The container builds and runs successfully, and when provided with a URL, the application scrapes the target site and serves the output via the web server.
- **Modularity & Clean Design:** Clear separation between the scraper stage and hosting stage, with minimal runtime overhead.
- **Documentation:** README clearly explains the setup, build, and run process.