Web Crawling Project for Python Developer

Problem Statement

You work for a content aggregation company that monitors product-related websites. Your current task is to crawl, monitor, and serve the data from a sample e-commerce website: **https://books.toscrape.com**. While this site is a simulation, treat it as production-grade.

Your responsibilities include:

1. Designing and implementing a scalable and fault-tolerant web crawling solution.
2. Implementing a change detection mechanism to maintain up-to-date records.
3. Creating RESTful APIs with authentication and filtering capabilities.
4. Ensuring the code is production-ready, with proper configuration, logging, and test coverage.

Part 1: Crawler with Robustness & Scalability

Goals:

- Crawl all book-related information and store it in a NoSQL DB (MongoDB preferred).
- Handle pagination, transient errors, and unexpected content structures.
- Store metadata (crawl timestamp, status, source URL) with each book.

Book data to collect:

- Name of the book
- Description of the book
- Book category
- Book prices (Including and Excluding taxes)
- Availability

- Number of reviews
- The image URL of the book cover
- Rating of the book

Additional requirements:

- Implement retry logic.
- Support resuming from the last successful crawl in case of failure.
- Use async programming (e.g., httpx, aiohttp) to speed up the crawl.
- Implement a Book schema with Pydantic.
- Design your MongoDB schema to support efficient querying and deduplication.
- Store raw HTML snapshots of each book page as a fallback.

Part 2: Scheduler and Change Detection

Implement a daily scheduler that:

- Detects newly added books and inserts them into the DB.
- Compares stored book data with the current site and updates changed records (e.g., price or availability changes).
- Maintains a change log in the database showing what was updated and when.

Additional requirements:

- Use a scheduling framework like APScheduler, Celery + Beat, or cron via Docker.
- Optimize detection by using content hash comparison or a fingerprinting strategy.
- Implement logging and alerting (e.g., via email or log file) when new books or significant changes are detected.
- Provide an option to generate a daily change report (JSON/CSV).

Part 3: Secure RESTful API Server

Build a RESTful API using FastAPI:

Required Endpoints:

1. GET /books
    a. Query Params:
        i. category
        ii. min_price, max_price
        iii. rating
        iv. sort_by (rating, price, reviews)
    b. Pagination support is required.
2. GET /books/{book_id}
    a. Return full details about the book.
3. GET /changes
    a. View recent updates (e.g., price changed; new book added).

Additional API requirements:

- Implement API key-based authentication.
- Add rate limitations (e.g., 100 requests per hour).
- Provide OpenAPI/Swagger documentation via the API.

Deliverables

- Push your code to a public Git repository (GitHub, GitLab, Bitbucket).
- Include:
    - Well-structured README.md
        - Setup instructions
        - Python version and dependency versions
        - Example .env file for config
    - Folder structure should separate:
        - Crawler
        - Scheduler
        - API
        - Utilities
        - Tests
    - Postman collection or Swagger UI for API testing
    - Sample MongoDB document structure
    - Screenshot(s) or logs of successful crawl + scheduler runs

Evaluation Criteria

- Code cleanliness, modularity, and readability
- Handling of edge cases and robustness
- Efficiency and scalability of crawling logic
- Clarity and structure of documentation
- Use of advanced Python features (asyncio, Pydantic, decorators, etc.)
- Security (API key handling, rate limiting)
- Test coverage

If you find anything confusing or have any question, don't hesitate to contact