Web Scraping Project Documentation

Overview:

The project consists of 4 core elements: Crawler, Scheduler, Apis & Tests.
Crawler: It is a program written in python which crawls a website (https://books.toscrape.com) using the popular scraping library beautifulsoup.
Scheduler: It is a python program that runs continuously and runs the crawler after a certain time to find any new or modified data in the website. If it finds any the database (DB) is updated accordingly, and json and csv reports are generated after the scheduler has finished crawling the entire website.
Apis: They are codes written in python using FastApi to query and retrieve book data stored in the DB.
Tests: They are used to test the apis and the crawler.

- Incode documentation is also provided for better understanding.

Primary Technologies and Libraries used:

Crawler: beautifulsoup
Scheduler: apscheduler
Apis: FastAPI
Tests: pytest
Database: MongoDB (MongoDB Compass)

Detailed Sections of The Project

Crawler:

This module contains an asynchronous web crawler designed to scrape **books.toscrape.com** (or any similar site) and store book metadata, snapshots, and change logs into a MongoDB database.

It handles:

- Fetching all paginated book links

- Scraping book details
- Detecting content changes via hashing
- Storing snapshots only when content changes
- Parallel crawling with concurrency limits
- Returning a list of all changes detected on a run

Environment Variables

- **BASE_URL** – Base website to crawl
- **CRAWL_CONCURRENCY** – Number of simultaneous requests
- **CRAWL_RETRIES** – Number of retry attempts for failed requests

Defaults:

- BASE_URL = "https://books.toscrape.com"
- CONCURRENCY = 10
- RETRIES = 3

Main Class: Crawler

__init__(base_url, concurrency)

Initializes the crawler with:

- Base URL
- Async HTTP client
- Semaphore to limit concurrency

async fetch(url)

Fetches HTML from a URL with:

- Concurrency limit
- Automatic retries
- Error logging

Returns: **HTML string**

async get_all_book_links()

Scrapes *every page* of the catalog to collect all book detail URLs.

Workflow:

1. Start at page 1
2. Extract <a> links for all books
3. Follow the **Next** button until pages end
4. Deduplicate links while keeping order

Returns: **List of book detail URLs**

parse_rating(soup)

Extracts star rating based on CSS class:
"One" → 1, "Two" → 2, …

Returns: **Integer or None**

parse_book(html, url)

Parses a single book page and extracts:

- Title
- Description
- Category
- Prices (incl/excl tax)
- Availability
- Number of reviews
- Image URL
- Rating
- Unique book ID (derived from URL path)
- Timestamp
- Content hash

Returns:
(book_data_dict, raw_html)

async process_book(url)

Processes a single book.

Behavior:

If book exists in DB

- Compares stored hash vs. scraped hash
- If changed:
    - Create snapshot
    - Update book record
    - Insert change log
    - Return change info
- If unchanged:
    - No snapshot created
    - Returns None

If book is new

- Create snapshot
- Insert book
- Insert change log
- Return change info

Returns: **Change record or None**

async run(resume=False)

Main crawling workflow:

1. Get all book links
2. Process each book concurrently
3. Collect change records

Returns: **List of changes detected this run**

Helper Functions

async crawl_all_books()

Utility for schedulers:

1. Runs the crawler
2. Fetches all books from DB (without raw HTML)
3. Returns them as a list

async main()

Simple script entry point:

```
asyncio.run(c.run())
```

Summary of Data Saved

Books Collection

- Metadata fields (title, category, price, etc.)
- content_hash
- raw_snapshot_id (link to snapshots collection)

Snapshots Collection

- Full HTML
- Timestamp
- Source URL

Change Log Collection

- book_id
- change_type (new / updated)
- timestamps
- old vs new hash (for updates)

Book Model Using Pydantic

```
class Book(BaseModel):
    id: str = Field(..., description="Unique id")
    title: str
    description: Optional[str]
    category: Optional[str]
    price_including_tax: Optional[float]
    price_excluding_tax: Optional[float]
    availability: Optional[str]
    num_reviews: Optional[int]
    image_url: Optional[HttpUrl]
    rating: Optional[int]  # 1-5
    source_url: HttpUrl
    crawl_timestamp: datetime
    content_hash: str
    raw_snapshot_id: Optional[str]
```

Schedular:

This module manages **automated crawling** of the book website using APScheduler. It triggers the crawler periodically, detects changes, and generates a daily report from those changes.

Logging Setup

A logger named "scheduler" is configured to print timestamped logs to the console to help track job execution and errors.

Function: scheduled_crawl()

An asynchronous task executed by the APScheduler.

Workflow:

1. Logs start of scheduled crawl
2. Creates a Crawler instance
3. Runs the crawler using c.run()
    - Returns a list of **new or updated books** detected during this run
4. Logs how many changes were detected
5. Calls generate_daily_report(changes)
    - Generates a report *only* from this run's changes
6. Closes the crawler's HTTP client

Returns:

Nothing, but generates a report and updates the DB.

Function: async_main()

Sets up and starts the APScheduler.

Behavior:

- Creates an AsyncIOScheduler for async tasks

Adds a job:
scheduler.add_job(scheduled_crawl, "interval", minutes=15)

- → Runs scheduled_crawl() **every 15 minutes**
- Starts the scheduler
- Keeps the program alive indefinitely using await asyncio.Event().wait()

Script Entry Point

If the file is executed directly:

asyncio.run(async_main())

This launches the scheduler, which will then execute the crawl job every 15 minutes.

Summary

- scheduled_crawl()
  → Runs crawler → gets changes → generates report
- async_main()
  → Starts APScheduler → schedules the job → keeps program running
- APScheduler interval: **15 minutes**

Alert & Report:

These modules are responsible for creating and sending reports respectively, after the scheduler crawls the entire website at a defined time interval.

Alert

The alert.py file is responsible for sending an email with the generated reports after each schedular crawl. For testing purposes, an email is sent regardless of whether there are any changes or not.

Report

The report.py file generates csv & json reports every time the scheduler crawls the website. The files contain any change in data of the website or no change message.

APIs:

This module defines the **FastAPI backend** for the Book Scraper system.
It exposes endpoints to retrieve books, book details, and change logs.
It also enforces API key authentication, rate limiting, pagination, and optional filtering.

App Initialization

- Loads environment variables (e.g., API_PORT)
- Creates a FastAPI app with metadata
- Registers global rate limiting via SlowAPI
- Enables CORS for all origins (useful for frontend access)

API Key Authentication

All endpoints include:

dependencies=[Depends(get_api_key)]

Only clients providing a valid API key can access the API.

Rate Limiting

Each route uses:

@limiter.limit("X/hour")

Examples:

- /books: **100 requests/hour**
- /books/{book_id}: **5 requests/hour**
- /changes: **100 requests/hour**

This protects the API from abuse.

book_doc_to_resp(doc)

A helper function that extracts only the allowed fields from a MongoDB document before returning it to the client.

It prevents leaking internal fields like raw_snapshot_id or content_hash.

Endpoint: GET /books

Returns a **paginated list of books**, with optional filters and sorting.

Supported Query Parameters:

- category — filter by book category
- min_price, max_price — price range filter
- rating — filter by rating (1–5)
- sort_by — one of:
    - "rating" (desc)
    - "price" (asc)
    - "reviews" (desc)
- page, page_size — pagination controls

Response Body:

```
{
  "page": 1,
  "page_size": 20,
  "total": 1000,
  "results": [ ...list of books... ]
}
```

The query is translated into a MongoDB filter, sorted if needed, then paginated manually using skip() and limit().

Endpoint: GET /books/{book_id}

Returns **one book** by its ID.

- Looks up _id in MongoDB
- If not found → returns 404 Book not found
- Otherwise → returns sanitized book document

Endpoint: GET /changes

Returns recent changes (new or updated books) from the change_log collection.

Details:

- Sorted by newest first

- Limited by query parameter limit (default 100)
- Converts _id and book_id to strings for JSON compatibility

Response:


```
{
  "results": [
    {
      "_id": "log123",
      "book_id": "bk021",
      "change_type": "updated",
      "timestamp": "...",
      "old_hash": "...",
      "new_hash": "..."
    }
  ]
}
```


Running the API

Use the following command:

uvicorn api.main:app --host 0.0.0.0 --port <API_PORT> --reload

Rate Limiter:

This module sets up **rate limiting** for the FastAPI application using **SlowAPI**.

limiter

limiter = Limiter(key_func=get_remote_address)

- Creates a global Limiter instance
- key_func=get_remote_address → limits requests per client IP address
- Can be used as a decorator on FastAPI routes to enforce rate limits, e.g.:

@limiter.limit("100/hour")

register_rate_limit(app: FastAPI)

Registers the Limiter with the FastAPI app.

Actions:

1. Attaches the limiter to app.state.limiter
2. Registers a **429 Too Many Requests** exception handler to return a proper response when a client exceeds the limit.

Usage:

```
from fastapi import FastAPI
from api.rate_limit import register_rate_limit

app = FastAPI()
register_rate_limit(app)
```

After this, any route decorated with @limiter.limit() is enforced, and exceeding clients get a 429 response.

Summary

- Provides a **global rate limiter** for the API
- Limits are **per client IP**
- Handles **429 errors gracefully**
- Works seamlessly with FastAPI decorators

Tests:

conftest.py

This module provides **fixtures and fake database classes** for testing the FastAPI Books Scraper API without requiring a real MongoDB instance. It enables **unit and integration tests** of API endpoints in isolation.

Key Components

1. Fake MongoDB Classes

- **FakeCursor**
  Simulates a Motor cursor with support for sort(), skip(), limit(), and to_list().
- **FakeCollection**
  Simulates a MongoDB collection with basic methods:

    - find_one(), find()
    - insert_one(), update_one()
    - count_documents()
- Supports simple filtering, range queries (e.g., $gte, $lte), and sorting.
- **FakeDB**
  Contains two collections:
    - books → stores book documents
    - change_log → stores book change records

These classes allow testing database-dependent API logic without a real database.

2. Pytest Fixtures

- **sample_books** → Returns a list of sample book documents for testing.
- **fake_db** → Returns a FakeDB instance preloaded with sample books and change logs.
- **client** → Provides an AsyncClient for testing the FastAPI app with:
    - get_db patched to return fake_db
    - API key dependency overridden to allow "testapikey"
    - Uses ASGITransport to make requests to the app in-memory

Purpose

This setup allows writing **async API tests** that:

- Query books with filters, sorting, and pagination
- Retrieve individual book details
- Fetch change logs
- Test authentication and rate-limiting logic

All without touching a real database or network.

test_api.py

This module contains **asynchronous pytest tests** for the FastAPI Books Scraper API.
It uses the client fixture (with a fake database and overridden API key) to test endpoints **without a real database or network**.

Tested Endpoints

1. **GET /books** – List books with optional filters, sorting, and pagination
2. **GET /books/{book_id}** – Retrieve a single book by ID
3. **GET /changes** – Retrieve recent changes (new or updated books)
4. **Authentication and rate limiting**

Tests Overview

Test	Purpose
test_list_books_no_filters	Returns all books when no filters are applied
test_list_books_category_filter	Returns only books in a specific category
test_list_books_price_range	Returns books within a specified price range
test_list_books_sorting_and_pagination	Validates sorting by rating and pagination
test_get_book_by_id_found	Returns correct book when ID exists
test_get_book_by_id_not_found	Returns 404 when book ID does not exist
test_changes_endpoint_default_limit	Returns all changes with default limit
test_changes_endpoint_with_limit	Returns only the specified number of change entries
test_unauthorized_access	Verifies 401 Unauthorized when API key is missing
test_validation_error_for_bad_query_param	Verifies 422 response for invalid query parameters
test_rate_limit_hit	Simulates rate limiting, expecting 429 if limit is exceeded

Key Features

- Uses **async tests** (@pytest.mark.asyncio) with AsyncClient
- Validates **response status codes**, payload structure, and content
- Covers **filtering, sorting, pagination, and authentication**
- Tests **rate limiting behavior** using SlowAPI configuration
- Fully isolated with **fake database and API key override**

Summary

These tests ensure that:

1. API endpoints behave correctly with various query parameters.
2. Authentication and rate limiting are enforced.
3. Edge cases (missing book, invalid input, query limits) are handled gracefully.

The test suite can be run with:
pytest tests/test_api.py

test_crawler.py

This module contains **asynchronous tests** for the Crawler class in the Books Scraper system.

Tested Functionality

get_all_book_links()

- Ensures the crawler correctly **collects all book detail URLs** from paginated catalogue pages.
- Uses **monkeypatching** to simulate HTTP responses, avoiding actual network requests.

Test Overview

Test	Purpose
test_get_all_book_links_monkeypatched	Simulates two catalogue pages with next links and multiple books. Verifies that Crawler.get_all_book_links(): • Follows pagination • Extracts all book URLs • Preserves correct order

Key Features

- **monkeypatch** replaces Crawler.fetch with fake_fetch returning predefined HTML.
- Tests **HTML parsing, pagination, and URL joining** without external dependencies.
- Verifies that the crawler correctly returns all expected links.

Summary

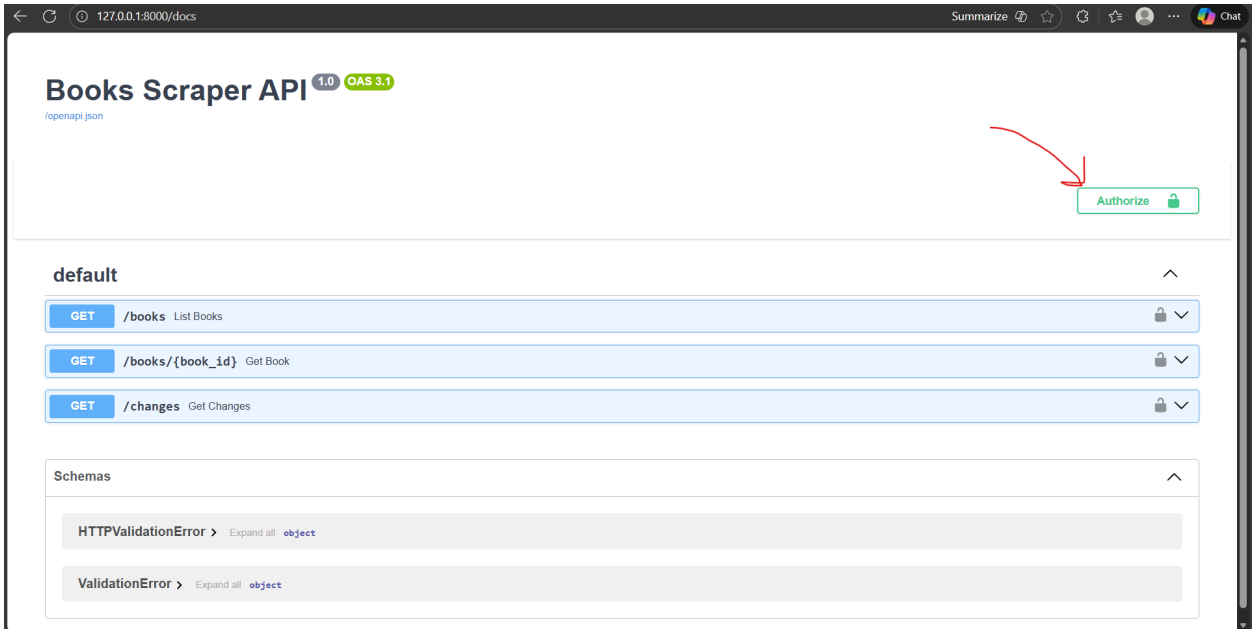
This test ensures the crawler:

1. Handles multi-page catalogues.
2. Extracts book URLs from the HTML correctly.
3. Returns a deduplicated, ordered list of links.

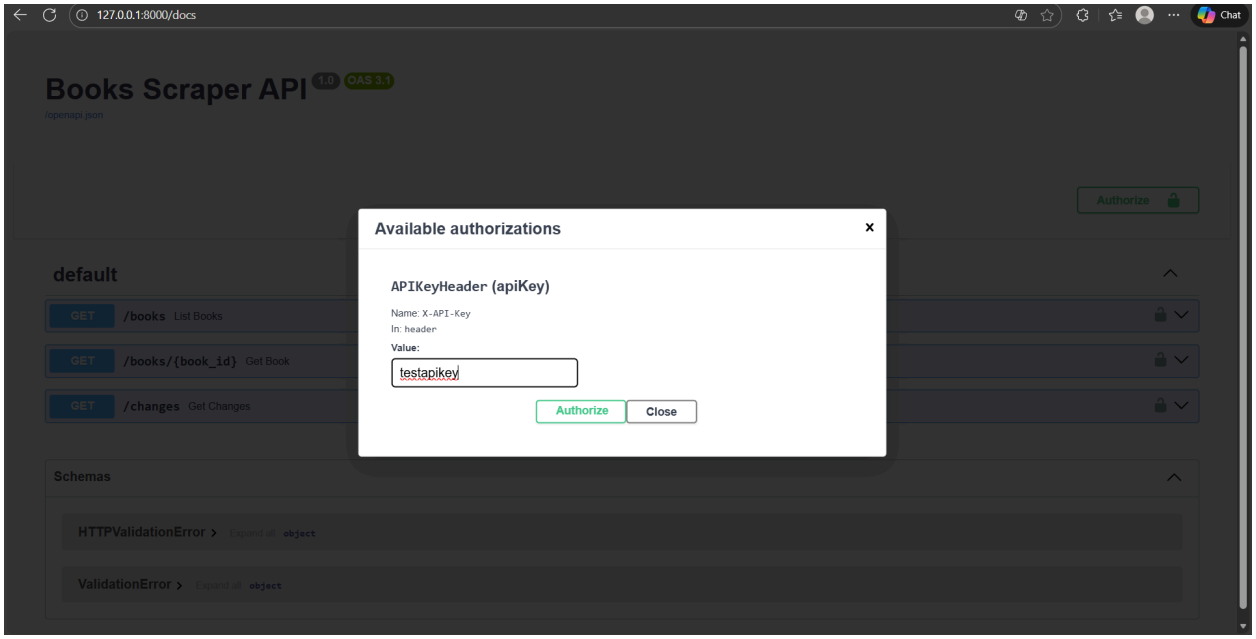
Swagger UI for API testing:

- To use the UI for testing go to the following url (after turning on the server): http://127.0.0.1:8000/docs

- After going to the UI click on the authorize button to enter the authorization header.



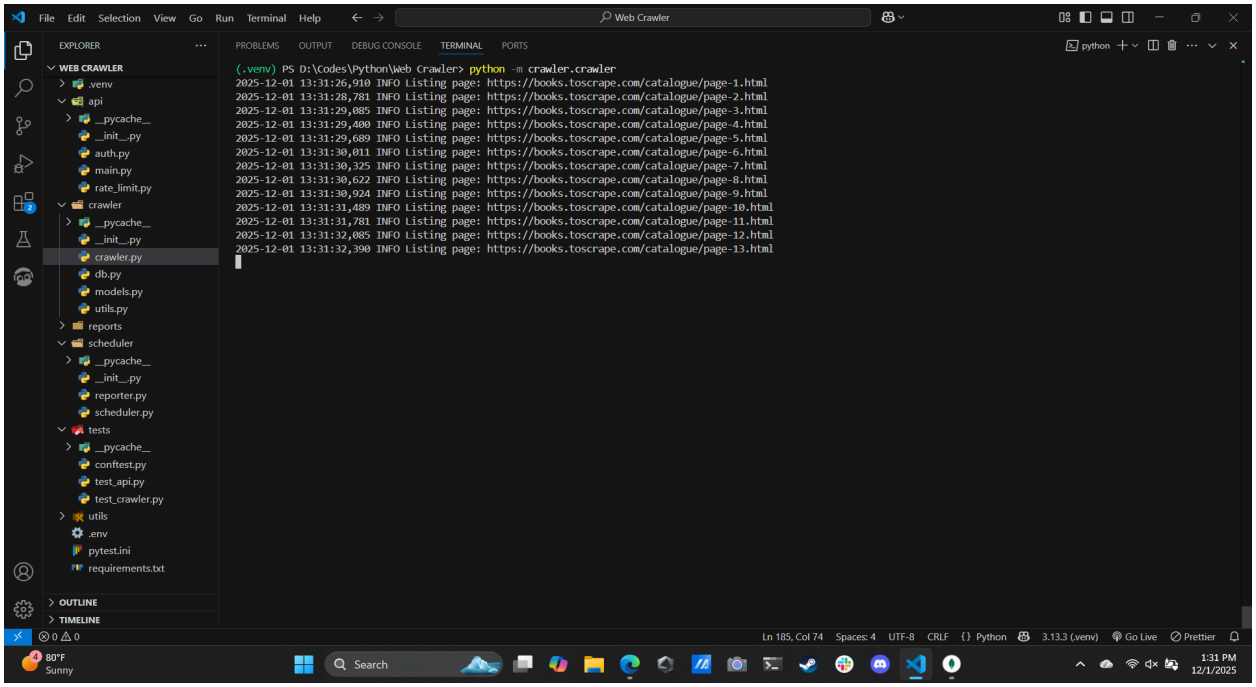
- Then input the value “testapikey” as the authentication header value like in the picture below:



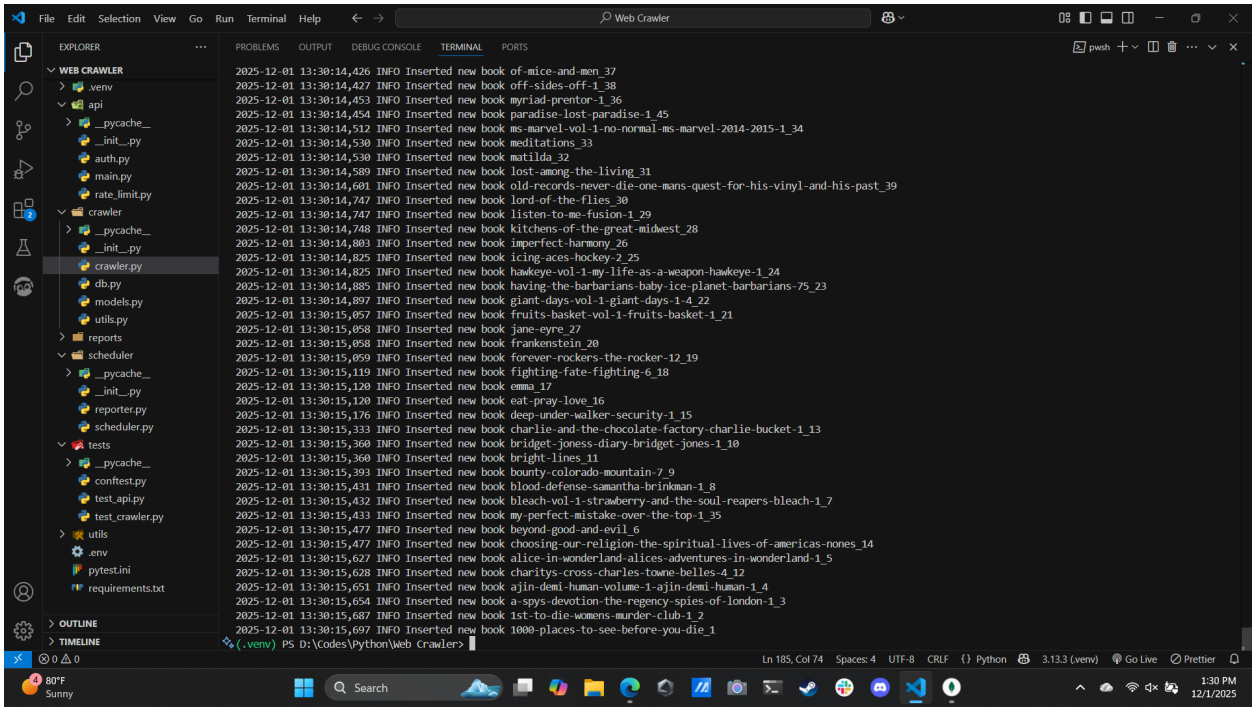
- Now you can test the APIs.

Important Screenshots:

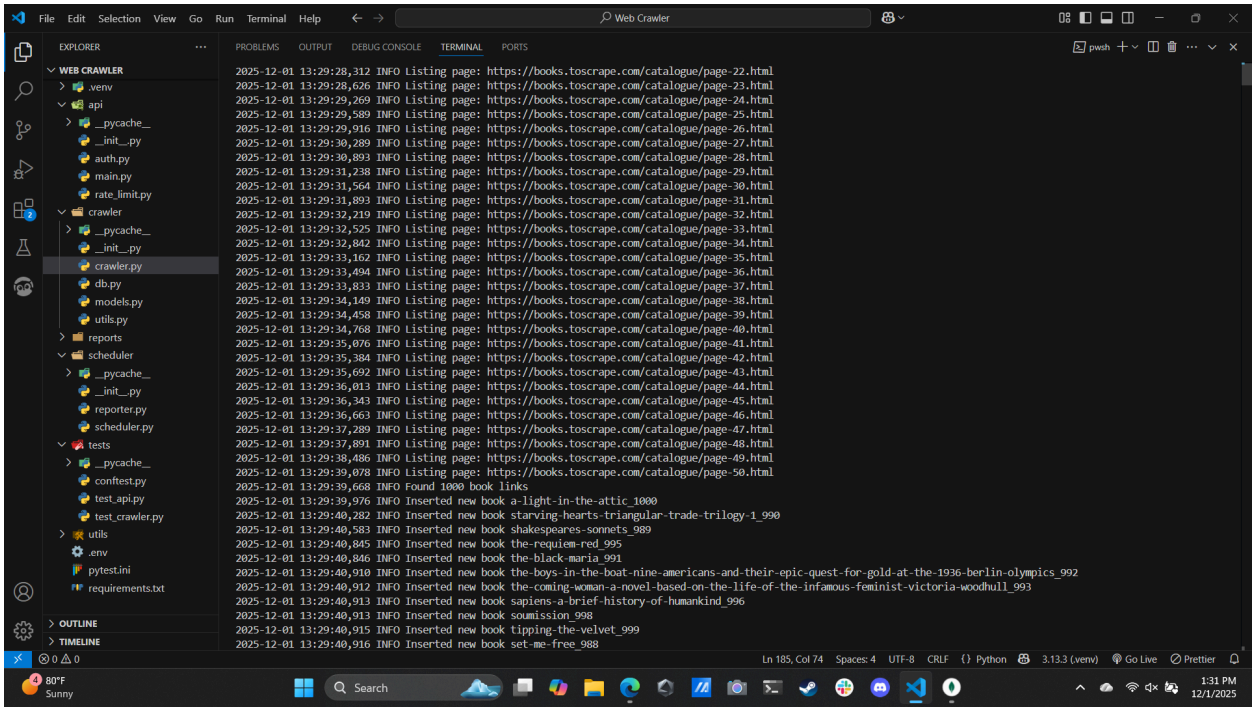
1) Crawler starting to crawl the website.



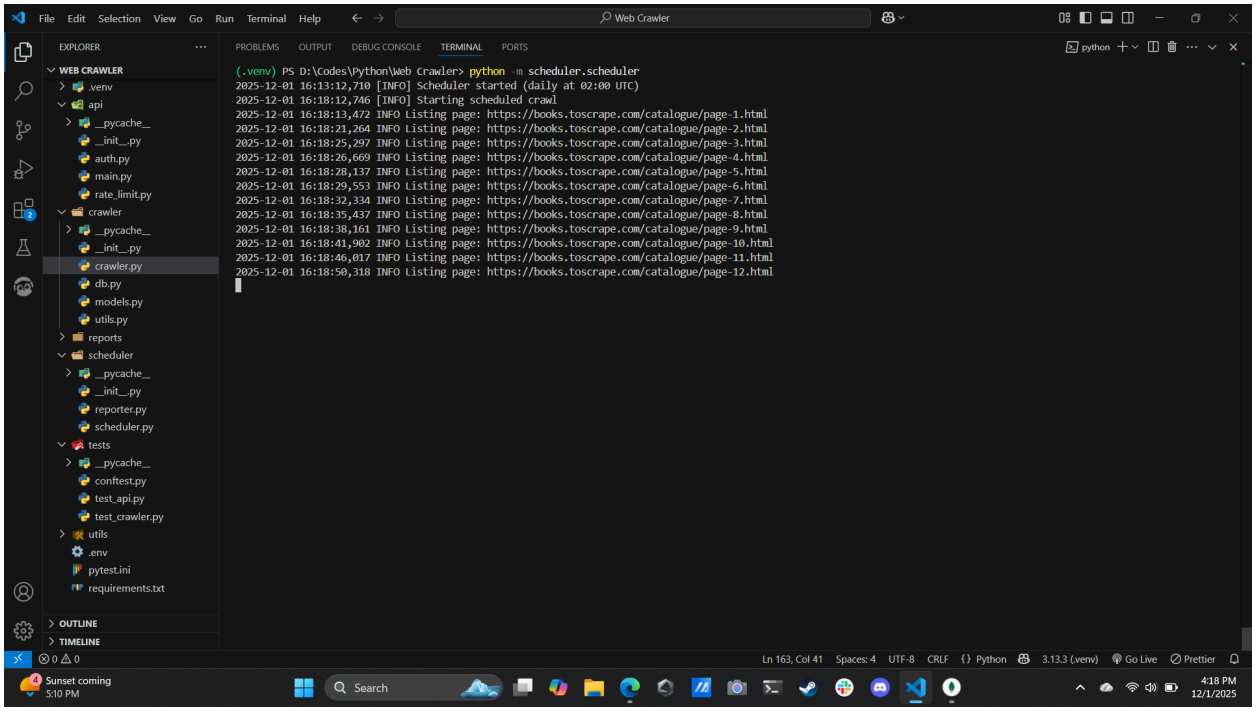
2) Crawler inserting books into MongoDB.



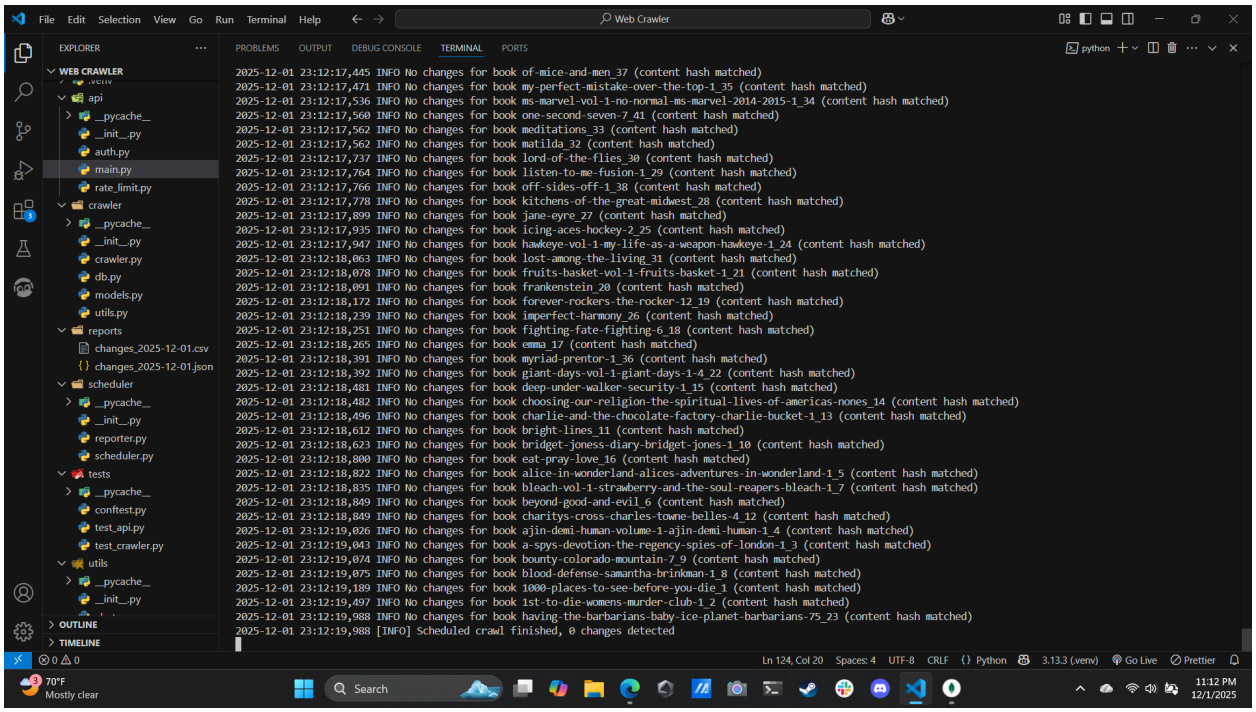
3) Crawler crawled the website and started to insert books into MongoDB.

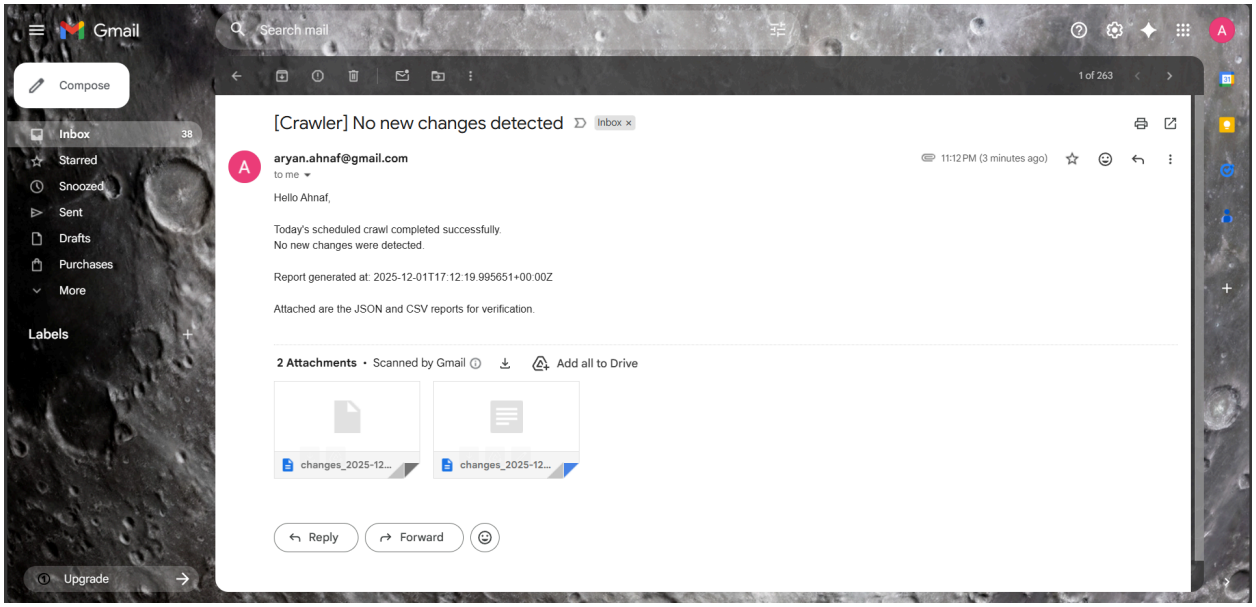


4) Scheduler starting to crawl.



5) The scheduler finished crawling and comparing the hashes of each book to detect any changes and then generated reports and sent an email alert.





6) All 12 tests passed.