Infosys Springboard Internship 5.0

Introduction:

Deals Hunter, Libraries Near You, and Jobs Scraper are Python-based projects that utilize web scraping to gather data from e-commerce, library, and job listing websites. Built with Streamlit, each project offers an interactive interface for users to easily access the latest deals, find nearby libraries, and explore job openings, all with real-time updates and customizable filters for a seamless experience.

Deals Hunter: Deals Hunter automates deal discovery on Deals Heaven, allowing users to search by store, category, and page range. This project saves time by streamlining the browsing process and enhances accessibility with an intuitive, user-friendly interface for easy deal access.

Libraries Near You: Libraries Near You simplifies finding public libraries by scraping data from the Public Libraries website. Users can search by state or library name, gaining quick access to detailed information, including addresses, all through a convenient and user-friendly interface.

Jobs Scraper: Jobs Scraper automates job searches on Behance Jobs, offering flexibility in search criteria with filtering options. It saves time by dynamically collecting relevant job listings, presenting them in an interactive UI, and enhancing the experience with light/dark theme options.

Project Scope:

Deals Hunter:

- **Included:** Scraping deals, filtering by store and category, storing in CSV, and displaying deals in a visually appealing UI.
- **Constraints:** Limited to the Deals Heaven website; functionality depends on stable internet and website structure.

Libraries Near You:

- **Included:** Scraping states and libraries, storing in relational tables, displaying data in the UI, and allowing CSV downloads.
- **Constraints:** Relies on pre-populated databases for efficient search; scraping the entire dataset is time-intensive.

Jobs Scraper:

- **Included:** Scraping jobs based on category and name, filtering by company, showing job details, and supporting light/dark themes.
- **Constraints:** Limited scrolling; users need to adjust scrolling limits in the code for larger searches.

Requirements:

Deals Hunter:

- Functional Requirements:
  - Scraping Module: Fetches deals based on user input (store, category, page range).
  - Data Storage Module: Saves scraped data into a CSV file.
  - Filtering Module: Filters deals by store and category.
  - Display Module: Displays deals in a visually appealing UI.
  - Progress Module: Tracks and displays the scraping process.
  - Error Handling Module: Manages errors during scraping and database operations.
- Non-Functional Requirements: Progress bar and user-friendly design.

Libraries Near You:

- Functional Requirements:
  - Scraping Module: Collects states and library information from the website.
  - Database Module: Manages relational tables for states and libraries.
  - Search Module: Allows users to search libraries by state or name.
  - Display Module: Presents data in a tabular format with download options.
  - Error Handling Module: Manages errors during scraping and database operations.
- Non-Functional Requirements: Optimized database queries and responsive UI.

Jobs Scraper:

- Functional Requirements:
  - Scraping Module: Extracts job listings based on category and name.
  - Data Storage Module: Temporarily holds scraped job data.
  - Filtering Module: Filters jobs by company using a dynamic search bar to fetch input.
  - Display Module: Displays job listings and supports light/dark themes.
  - Error Handling Module: Captures and logs errors during scraping.
  - Theme Module: Enhances user experience by allowing theme switching.
- Non-Functional Requirements: Flexible search functionality and improved UI through theme support.

User Stories/Use Cases:

1. As a user, I want to input search parameters (such as job title, deal category, or library location) and initiate a scraping process so I can obtain relevant data for my needs.
2. As a user, I want to set limits for searching (such as page range and scroll limit) to save time and make the scraping process more efficient.
3. As a user, I want to view the scraped data in a clear, organized format within the UI so that I can easily browse and analyse the results.
4. As a user, I want the scraped data to be saved as a CSV file so I can store it for future reference or share it with others.
5. As a user, I want to see real-time progress feedback during the scraping process so that I know when the data is ready to be viewed.
6. As a user, I want to not only see the deals but also access the deals directly from the displayed data.

Technical Stack:

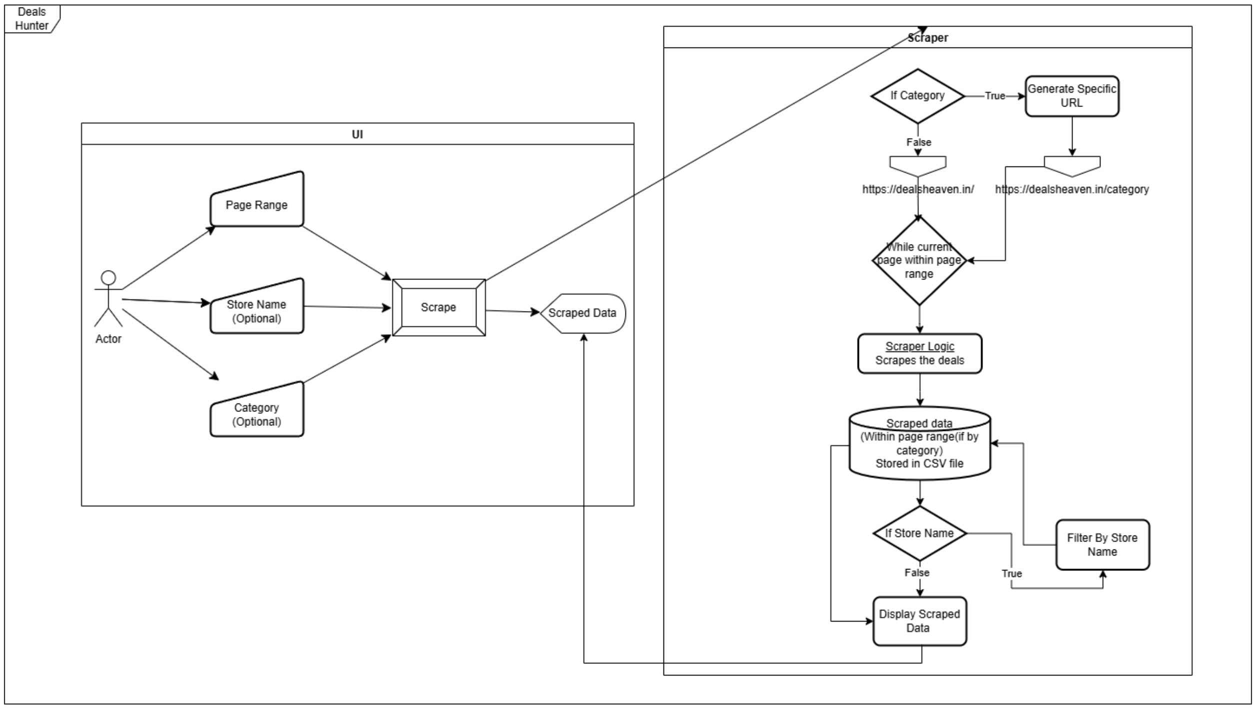
Programming Languages: Python
Frameworks/Libraries: Streamlit, BeautifulSoup, Selenium, Pandas, CSV, Requests, Time
Databases: SQLite (Libraries Near You)
Tools/Platforms: Visual Studio Code, Web Browsers (Chrome/Edge)
Version Control: GitHub

Architecture/Design:

Deals Hunter:

Scraper module fetches deals, stores them in a CSV, and displays them in the UI with filtering options.

The interaction diagram shows the flow from user input to scraping, storage, filtering, and display.

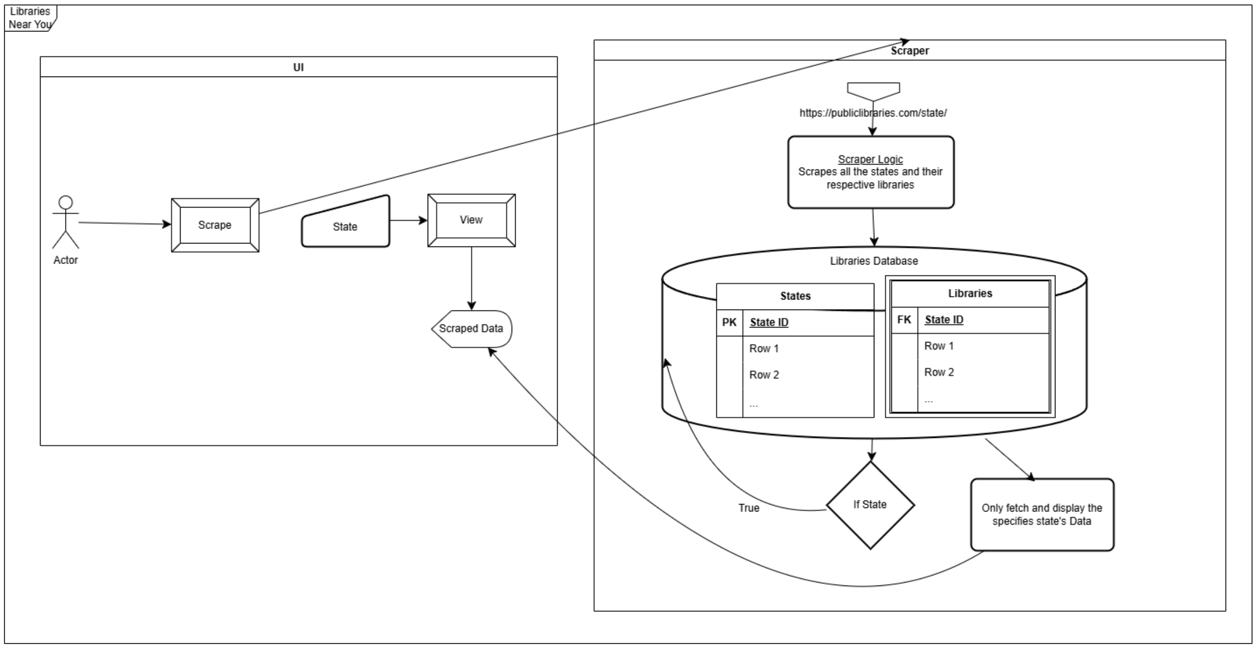


Trade Offs: Using **BeautifulSoup** for scraping was faster and more efficient for static websites, but it couldn't handle dynamic JavaScript content as well as **Selenium**

Libraries Near You:

Scrapes states and libraries, stores data in relational tables, and provides a UI for search and display.

The interaction diagram shows the flow from user input to scraping, storage, filtering and display, including the relation between the databases.

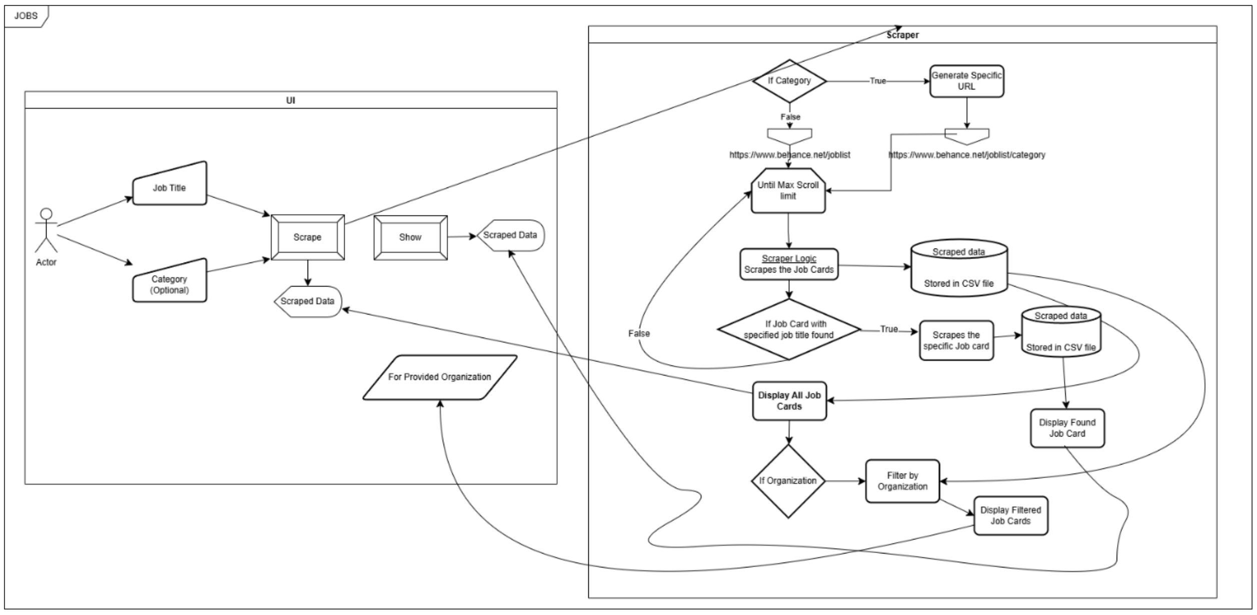


Trade-Offs: Used SQLite for ease of integration and management of relational data.

Jobs Scraper:

Automates job search with specified parameters, stores results temporarily, and displays filtered results in the UI.

The interaction diagram shows the flow from user input to scraping, storage, filtering and display, including the relation between the databases.



Trade-Offs: Implemented Selenium for dynamic scrolling and interaction capabilities.

Development:

The project adhered to coding standards and best practices such as modular design, PEP 8 compliance, robust error handling, secure input validation, version control with Git, and streamlined dependency management.

DealsHunter:

- **Technologies:** Built using BeautifulSoup for scraping and Streamlit for the UI.
- **Process:** Developed in agile milestones to integrate features like filtering and a progress tracker step-by-step.
- **Challenges:** Handled dynamic website structures and CAPTCHA-related restrictions.
- **Solutions:** Introduced modular scraping logic and fallback mechanisms for handling website changes.

Libraries Near You:

- **Technologies:** Selenium for scraping and SQLite for relational data management.
- **Process:** Built relational tables for scalability and integrated Streamlit for efficient data display.
- **Challenges:** Encountered performance degradation when handling large datasets.
- **Solutions:** Optimized database queries, indexed key columns in SQLite, and implemented lazy loading for data display to improve performance.

Jobs Scraper:

- **Technologies:** Selenium for dynamic scrolling and Streamlit for an interactive UI.
- **Process:** Automated job scraping and added features like dynamic filtering and theme switching.
- **Challenges:** Encountered difficulties with changing website layouts and anti-scraping mechanisms.
- **Solutions:** Enhanced error handling and implemented a flexible scraping framework.

Testing:

Approach:

1. Unit Testing:
    - Verified individual components like the scraping module, data processing logic, and UI rendering.
    - Ensured each module operates correctly in isolation.
2. Integration Testing:
    - Assessed seamless interaction between backend scrapers, databases, and frontend UIs.
    - Confirmed that data flows correctly from scraping to storage and display.

3. System Testing:
    - Validated the overall system functionality under various user scenarios and datasets.
    - Ensured system stability, responsiveness, and error-free operations across the application.

Results

1. DealsHunter:
    - Delivered precise and consistent deal scraping.
    - UI rendered accurately with minimal latency during testing.
    - Fixed an issue where incorrect category inputs caused errors in scraping logic.
2. Libraries Near You:
    - Achieved reliable data retrieval and search functionality for state and library data.
    - Resolved a bug where large datasets caused database query timeouts by implementing indexing.
3. Jobs Scraper:
    - Enabled smooth job listing retrieval with dynamic filtering and light/dark theme support.
    - Fixed bugs related to dynamic scrolling limits and unexpected UI crashes during invalid input.

Deployment:

The deployment process is streamlined for ease of use, leveraging automated scripts to maintain consistency and reducing manual setup errors. Applications are scalable to various hosting environments based on user needs.

Deployment Process:

1. All projects are deployable on **Streamlit Cloud**, providing easy accessibility and low setup overhead for end-users.
2. Deployment scripts automate the setup of dependencies and configurations, ensuring consistent environments for both scraping backends and UI frontends.
3. For advanced needs, the applications can be hosted on other platforms like Heroku or AWS with minimal adjustments.

Deployment Instructions:

1. Clone the Repository:
    - Use git clone < Github repo > to download the project.
2. Install Dependencies:
    - Navigate to the project directory and run:

      "pip install -r requirements.txt"
3. Run the Application Locally:
    - Execute:

      "streamlit run ui.py"

4. Deploy on Streamlit Cloud:
    - Push the repository to a connected GitHub account.
    - Navigate to Streamlit Cloud, create a new app, and select the repository.
    - Specify app.py as the entry point and deploy.
5. Optional Hosting Options:
    - For platforms like Heroku or AWS, use a Dockerfile or a Procfile for deployment. Adjust configurations as needed based on the target platform.

User Guide:

Deals Hunter:

Setup and Configuration:

1. Install dependencies:
    - Install the required libraries by running the following command:

      “pip install -r requirements.txt”
2. Run the application:
    - After installing dependencies, run the application with the following command:

      “streamlit run ui.py”
3. Input Parameters:
    - When prompted, enter the following:
        - Store: Name of the store you wish to scrape deals from.
        - Category: Category of deals you're interested in.
        - Page Range: The range of pages you want to scrape deals from (e.g., 1-10).

Troubleshooting Tips:

- Error: "No deals found":
    - Ensure the store name and category are correctly typed and available on the source website.
    - Check that the website is up and running.
- Error: "Timeout error":
    - If scraping takes too long, adjust the number of pages or check your internet connection.

Libraries Near You:

Setup and Configuration:

1. Pre-populate SQLite Database:
    - Before running the app, ensure the SQLite database with library data is pre-populated. You may need to populate it using scraper or use the provided database file.

2. Run the application:
    - Launch the application by running:

        “streamlit run ui.py”
3. Select a State or Search by Library:
    - In the app, choose a state to view libraries in that region or search for a library by name.

Troubleshooting Tips:

- Error: "No libraries found":
    - Ensure the database is correctly populated with state and library data.
    - Check if the state or library you are searching for is listed.
- Error: "Database connection issue":
    - Verify the database connection details in the app and ensure the SQLite database is accessible.

Jobs Scraper:

Setup and Configuration:

1. Install dependencies:
    - Install the required libraries:

        “pip install -r requirements.txt”
2. Ensure Web Driver Configuration:
    - Ensure that you have the appropriate Selenium WebDriver installed and configured on your machine. For Chrome, download the ChromeDriver and ensure it matches your Chrome version.
3. Run the application:
    - Launch the application with:

        “streamlit run ui.py”
4. Filter Jobs:
    - Enter the job category (e.g., software engineering) and job name (e.g., developer).
    - Use the dynamic filters to filter job results by company.

Troubleshooting Tips:

- Error: "Selenium WebDriver not found":
    - Ensure you have installed the correct WebDriver for your browser and that it is in your system’s PATH.
- Error: "Page load timeout":
    - If the webpage is slow to load, try adjusting the timeout settings in the Selenium code or check your internet speed.

- Error: "No job listings found":
  - Double-check the job category and company name for spelling errors.
  - Ensure the website you’re scraping from has job listings available.

General Troubleshooting Tips:

- Dependencies Issue:
  - If you encounter errors while installing dependencies, try upgrading pip:

    “python -m pip install --upgrade pip”
- Compatibility Issues:
  - Ensure that you are using a compatible version of Python (preferably Python 3.8 or above).
- Streamlit Errors:
  - If Streamlit fails to run, check that you have the correct Python version installed and that all dependencies are installed without errors.

Conclusion:

- **Deals Hunter:**
  - Enabled flexible deal searches by allowing users to scrape data based on store, category, and page range.
  - Gained practical experience with web scraping using Beautiful Soup, extracting deals from e-commerce websites.
  - Designed an intuitive UI with Streamlit, ensuring a seamless and interactive user experience for deal searches.

- **Libraries Near You:**
  - Worked with SQLite for relational database management, storing and retrieving library data effectively.
  - Utilized web scraping to gather and display library information based on state or name.
  - Created an interactive UI with Streamlit, allowing users to search and explore libraries easily.
- **Jobs Scraper:**
  - Automated scrolling and filtering of job listings, improving the scraping efficiency and user experience.
  - Explored Streamlit theme customization, enhancing the interface for a better user experience.
  - Developed a filtering system to allow job searches based on category and company.

Reflections on Lessons Learned and Areas for Improvement:

- **Lessons Learned:**
  - Web Scraping: Improved skills in scraping data using BeautifulSoup and Selenium, understanding how to extract and process information from websites.
  - UI/UX Design: Streamlit enabled the development of user-friendly and interactive applications, emphasizing simplicity and engagement.
  - Automation: Automated data scraping tasks, improving efficiency and scalability.
- **Areas for Improvement:**
  - Error Handling: More robust error handling would help manage edge cases like network issues or unexpected site changes.
  - Performance Optimization: Future work could focus on optimizing scraping algorithms and improving database efficiency for handling large datasets.
  - Testing: Writing more unit tests and conducting integration testing would ensure stability and minimize bugs across different modules.

In conclusion, these projects have honed my skills in web scraping, UI design, and automation while offering valuable insights into areas for further improvement, such as error handling, performance, and testing.

* Appendices:

Project Demo Video