Bobcat C.L.A.W.S.

Final Report

Group Members: Abigail De Rousselle, Dillon Hughes, Dustin Bruce, Garrett Dipalma, Jawad Abu-Gabal, Jess Stevenson, Kevin Garcia, Rayyan Khan, & Rebekah Hardsand

Date: 11/30/23

Contents:

Intro

This is our final report for the C.L.A.W.S. project. C.L.A.W.S. stands for College Lifestyle Affordable Wares Search, and we started this project with the purpose of helping students find the best price for the different necessities of college. The central idea of the project was to create a web page that would compile items from different storefronts and compare the prices. This document will summerize the work we have completed and where we think we have fallen short after just about 3 months of work.

We found the easiest way to divide the work was to make 3 distinced sections and then combine them towards the end of our time. Starting with the WebPage, the user facing compenet,

it shows the images, prices, and links to purchase for the different items we have compiled. Behind the WebPage is the database that funcions to maintain, sort, and organize the data sets for the WebPage. And finally we have the Server which stands behind the rest of our project hosting the WebPage, maintaining the database, and running our scrappers.

WebPage

Our webpage is styled after Texas State University, sporting the maroon and gold colors. It is made up of differnt components the work together to show the products we have gathered, their prices, and where to buy them. It also features a search bar and categories tabs to help users quickly find what they need.

Completed

- **Header:** The header component stays the same across all four main components of the application. The header shows 3 categorical dropdowns with their respective subcategories. The header also includes the Bobcat CLAWS logo where the user is able to click the logo to redirect them back to the homepage. The header also includes the search bar component where the user is able to search keywords to find needed products. Additionally we added an "ALL" tab where the user is able to view all categories offered by the application to make it easier for the user to navigate and see all options available for categories.
- **Homepage:** The homepage is the first point of interaction with the user. It displays 25 randomly selected products from the database to showcase a variety of products. The product page is displayed when the user clicks an item from the homepage or when the user clicks on an item in the categories page. The purpose of the products page is to be able to refresh when the user clicks an item and display the rest of the products list.
- **Categories Page:** When the user has selected a category they want to view from the header component, either the submenu dropdown or within the "ALL" tab dropdown, they are directed to the category page. The category page displays a list of all product data that is fetched from the database for the corresponding category. The user sees a list of product data that dispalys the name, price, a picture and a link to the products retailer.
- **Searchbar:** The search bar captures user input and on click submits user's query to the search service which allows search results page to gather results based on the passed down query.
- **Search Results Page:** The search results page is implemented so the user can search any keywords that are stored in the database. When the user enters a search term in the search bar of our header component they are redirected to the search results page. The search results are saved and sent to the database to fetch all products that include the respective keyword. The list of product data is then displayed to the search results page so the user can browse any products related to that search term. When the user clicks any of the products displayed on the search results page they are then directed to the products page.
- **Save Service:** The save service was implemented to interact with the backend side of the application. The service is responsible for fetching product related items from the database including fetching products by category id. The search service is in place to fetch product data from the database based off the users keyword search.The shared data service is in place so that the product array that is saved either from the search results page to the product page or the categories page to the product page. It is able to share the data list between components. The node server js file creates a web server that listens for specific web requests, executes a stored procedure on the MySQL database using theperamters provided in the request and returns results to the front end.

Incomplete

- **Homepage:** Planned on having coupons displayed on the homepage as "deals" section.
- **Categories:** Product page does not have any filtering. We could add a retailer filter, a price filter. All main components should include the retailer name. We can add a sort filter. Add all sort filters to a seperate component and get that working for all pages.
- **Searchbar:** We had planned on having filters implemented for the user to narrow the search for a desired product.
- **Search Results Page:** Users can not search on search results page for products, but can on every other page. Pagination needs to be added to the results gathered to allow easier interaction and viewing.

3. Database

We designed and implemented a MySQL database to efficiently store, manage, and serve product data to the front end. Our database has a data flow starting from categories, subcategories, stores, products, and then prices for each product depending on the store. We also have tables for coupon data depending on stores or products.

We have 3 categories and 16 subcategories. All data we currently have has been collected using Walmart and Target APIs and scrapping data from Best Buy. Products are categorized under one category and one subcategory.

The database tables are:

- Category
- Sub_category
- Product
- Store
- Product_Store_Price
- Store_location
- Coupon
- Coupon_Product
- Coupon_Store

Completed

- **Design and Setup:** The database is implemented using MySQL, running on the server's localhost, on the default port 3306.
- **User Access:** The database has user access for developers with full privileges, and for backup users with dump access privileges.
- **Insertion scripts:** Implemented scripts to insert product data from a formatted JSON containing the scrapped data or from APIs.
- **Readjusting design:** During our development we had to readjust some tables to better fit our data needs, some of these tables were the category, subcategory and product.
- **Populate database:** Used the insertion scripts to populate the database with actual final data obtained from scrapping and API calls.

- **Generate product keywords:** We have created a script to generate keywords for products in the database. These keywords are used for the search functionality on the front end.
- **Create stored procedures to send queries from front end:** Created stored procedures to get products based on subcategory IDs, category IDs, or all available products, and for the search functionality. These procedures are used by the front end to query the database for different pages.
- **Testing scripts and stored procedures:** Insertion scripts were tested to ensure data was stored in the correct tables, and ensure to not duplicate data for current products. Stored Procedures were tested to ensure the correct format and data was returned for each one.

Incomplete

- **Coupns:** Populating database with coupon data and creating procedures for front end to retrieve coupon data.

Server

The server is made up of multiple components that work together to supply various functions of our project. Starting with the base that serves as the background and hosting space for our project. It maintains storage and security for our webpage, programs, and database. The base of our server is formated as such: Client (HTTPS://labdev00.cs.txstate.edu) --> Nginx (port 443) --> |-- Reverse Proxy (to port 8080) --> Node.js Application (localhost:8080).

Completed

- **Operating System:** The server is running on Linux Mint, a popular Linux distribution known for its user-friendliness and elegance.
- **Web Server:** Nginx is configured as the main web server. It is responsible for handling client requests and serving web content.
- **Ports:** Nginx is listening on port 443, which is the standard port for HTTPS traffic. Node.js is set up to serve the webpage on localhost at port 8080.
- **Reverse Proxy:** Nginx is configured as a reverse proxy, which means it accepts incoming traffic on HTTPS port 443 and forwards it to the Node.js application on port 8080. This allows Nginx to handle the initial connection and perform tasks such as SSL termination and request caching before passing the request to the Node.js server.
- **SSL Certificate:** The server uses an SSL certificate issued by Texas State University ensuring that the communication between the client and the server is encrypted and secure. The presence of a valid SSL certificate also means that the setup is capable of handling HTTPS requests, which is critical for security, particularly if sensitive data is being transmitted.
- **Node.js Application:** The Node.js application serves the webpage from localhost on port 8080. This application is where your website's server-side logic runs, and it's where the content is generated or fetched before being served to the client.
- **Test Hosting:** Successfully set up and verified hosting capabilities, ensuring that our server can manage the intended traffic and loads effectively.
- **Backup Script for Database:** Implemented a reliable backup script to regularly save database content, ensuring data integrity and quick recovery in case of any data loss.
- **Schedule API Call Script:** Developed and deployed a script for automated API calls, facilitating consistent and efficient data exchange with external services.
- **Get HTTPS and Angular App to Show:** Successfully configured the server to serve the Angular app over HTTPS, enhancing security and user trust.

- **Register Domain**: Acquired a suitable domain name, establishing a professional web presence and making the application easily accessible to users.
- **Get Valid SSL Certificates**: Obtained SSL certificates to secure our website, to ensure encrypted connections and enhance user trust.
- **Schedule Node, Nginx, and Ng Build to Always Have Up-to-Date Angular App Running**: Implemented a system to automatically update and deploy the latest version of our Angular application, ensuring users always have access to the latest features and fixes.
- **Created Makefile**: Developed a Makefile to automate the compilation and deployment processes, significantly improving efficiency and reducing potential human errors.
- **Setup Firewall and Open Ports**: Configured the server's firewall and opened necessary ports, ensuring secure and smooth network communication for the services.
- **Fix Backend Node Errors**: Resolved issues in the Node.js backend, leading to improved stability and performance of server-side operations.
- **Setup Outside Txst Network Hosting**: Expanded our hosting capabilities beyond the Txst network, allowing for wider accessibility and reach.

Incomplete

- **Validate Security on Scripts**: Pending a thorough security review and validation for our scripts to ensure they are secure from vulnerabilities and do not pose any risk to our server or data.
- **Fix Makefile Errors**: Need to address and rectify errors in the Makefile to ensure smooth automation of our build and deployment processes. The current issues are hindering our ability to efficiently manage application updates and deployments.
- **Update Readme**: This task involves revising and enhancing the project's README file to provide up-to-date information. A well-maintained README is crucial as it serves as the first point of reference for new users or developers, offering guidance on installation, usage, and contributing to the project. The update may include details about recent changes, improved instructions, and additional documentation to reflect the current state of the project.
- **Update Git Repo Documentation**: This issue pertains to updating the documentation within the project's Git repository. Proper documentation in the repository is essential for effective team collaboration and clear communication of changes. This update should include detailed descriptions of the repository's structure, any new features or changes in the codebase, contribution guidelines, and possibly a changelog. Addressing this issue will help in maintaining transparency in the development process and assist new contributors in understanding and working with the project more effectively.

2. Data Collection

To gather the data that we needed to populate our database we turned to a few tools to help us. Starting with parseHub to actually scrape data off of store front pages we found that it could not gather all of the information that we wanted. We then started using the BlueCart and RedCircle APIs for Walmart and Target respecively. Eventuallly we began developing our own in house scrapper to meet the number of pulls and amount of information we needed.

2.1. ParseHub

Parsehub is a three part tool that allows a user to scrape data from a websites. The three parts of the tool are a project, an API, and a script, Parsehub.py. For Bobcat C.L.A.W.S. Parsehub was utilized to scrape product data from store websites that did not have their own API to automatically collect and provide that product data for them. The Parsehub program allows a user,

using a GUI interface and logical rules, to create a project that indicates what data is to be pulled from a type of webpage. The API, using that project, pulls the desired data from a given URL and returns it to a user. Parsehub.py calls on the API to initiate the project and return product data back to it. The script then searches for and corrects any missing data, formats the data, then sends the data directly to the database.

2.1.1. Completed

- **Create Parsehub Project:** Parsehub project was created using the Parsehub program's GUI interface that uses logical rules to scrape product data from search result and product detail pages. Scraped data includes a products URL address, UPC, Name, Price, Category, Subcategory, Description, and Image URL.
- **Add Pagination to the Parsehub Project:** A logical rule was inserted into the Parsehub project to click the next page button on the search results page after all product data on the current page was scraped, until there is no longer a next page button.
- **Create Parsehub Progmatic API Script:** A script, Parsehub.py, was created that calls on the API to initiate the scraping project on a given webstore's search results URL. For each URL in a list of URLs, leading to the search result pages for each desired type of product, the script calls on the Parsehub API to initiate the Parsehub scraping project. Upon receiving the scraped product data the script checks for missing data, corrects what it can. The data is then formatted into a form that is usable by the database and then collected into a list containing all product data collected from each URL. All collected data is then sent to the database.
- **Edit Script to Read from a searchTerms.txt File:** Parsehub.py was temporarily using a list of search terms, representing each desired product type to be scraped, that was hardcoded into the script. Instead, the Parsehub.py script was altered to read the from searchTerms.txt the list of search terms .
- **Edit Script to Create its Own List of URLs:** A function was added to the Parsehub.py that created the list of URLs leading to the search results page for each search term. The function adds each term to the end of a URL that references a search results page, for an undefined term, for a particular store.
- **Implement Error Handling:** Multiple checks and solutions were added to the Parsehub script to deal with potential errors in pulling the data. A retry mechanism was added to account for instances where the project was unable to connect to the API or a URL. The function check_values() was added to review the returned data and fill in any missing data that it can. In both cases if the script was unable to resolve the error on its own the script sends an email to the developer notifying them of what caused the error so that the developer may investigate and fix the error themselves.
- **Isolate Relevant Credentials:** Sensitive data, such as the API key and project tokens, were added to api.env. The script was altered to no longer have the credentials hardcoded into the script, instead it reads in the sensitive data from the .env file thus keeping the data private.
- **Output Data to Formatted JSON:** The Parsehub.py was altered to save all scraped and formatted product data into a JSON file that is saved in the json_files folder. The database is able to read the JSON data into their database to be used by the front end.
- **Implement Tests:** test_Parsehub.py was created to test the logic and functionality of all functions in Parsehub.py.
- **Update Parsehub.py:** As test_Parsehub.py was being created small changes to the formatting and logic of Parsehub.py were made. Changes were intended to make the script more concise and readable but not to alter the functionality of the script itself.

2.1.2. Incomplete

- **Scrape UPC, Description, Category and Subcategory Data:** The Parsehub program is unable to successfully scrape UPC, Description, Category, and Subcateogry product data. This appears to be due to Parsehub's own capabilites or lack thereof, and should be fixed as Parsehub program is further updated. In the mean time the UPC, Category, and Subcategory data was left as null, and the Description was set to the product Name. The missing data does not have a significant negative impact the functionality of the script, database, or frontend.
- **Create a File with URLs for Parsehub:** The Parsehub script was originally intended to read in search result URLs from a .txt file. These URLs were first going to be created manually by the developer but automation was a key value in creating Bobcat C.L.A.W.S., so the idea was scraped. Next the URLs were going to be automatically generated by saving the URLs of webstore categories and subcataegories that most closely resembled the desired search term. But it was found that the categories and subcategories were often either too general or too specific. This idea was scraped, and it was decided to generate URLs by concatenating a search term to an incomplete URL leading to a search results page of a specific webstore.
- **Rework Parsehub.py to Insert Directly in the Database:** It was decided it would be faster to have the Parsehub.py script send the JSON data directly to the database instead of a folder that the database would read from. Work is still being done to complete this issue.

2.2. BlueCart & RedCircle

BlueCart and RedCircle are the APIs for the Walmart and Target webstores respectively. The APIs pull product data from these webstores and return it to their respective scripts, wmAPI.py for Walmart and tAPI.py for Target. Both tAPI.py and wmAPI.py format the data and then sends the data directly to the database.

2.2.1. Completed

- **Targeted Product Scraping:** Script tAPI.py and wmAPI.py were created to use their respective APIs to pull product data from their respective webstores, format the returned data, and send the data to the database.
- **Add Array of Search Terms to wmAPI.py and tAPI.py:** An array of search terms were hardcoded into the wmAPI.py and tAPI.py to be supplied to their respective API that would search and return data on products that relate to each term.
- **Implement Error Handling:** An exception handling block was added to the code to catch any errors when the script tries to connect to the API.
- **Isolate Relevant Credentials:** The API key was added to api.env. The script was altered to no longer have the credentials hardcoded into the script, instead it reads in the sensitive data from the .env file thus keeping the data private.
- **Rework tAPI.py and wmAPI.py to Insert Directly in the Database:** It was decided that it would be faster to send all product data directly to the database. Both tAPI.py and wmAPI.py were altered to send the product JSON data directly to the database instead of a folder that the database would read from.
- **Implement Tests:** test_tAPI.py and test_wmAPI.py were created to test the logic and functionality of all functions in tAPI.py and wmAPI.py respectively. Both tAPI.py and wmAPI.py function the similarly so the tests for both scripts are mostly the same.

2.2.2. Incomplete

- **Pagination in API Tool:** Pagination was not implemented in tAPI.py and wmAPI.py as their respective APIs both handle collecting all product data related to each search term.
- **Output Data to Formatted JSON:** The tAPI.py and wmAPI.py were originally intended to save product data JSON was altered to save all scraped and formatted product data into a JSON file that is saved in the json_files folder. The database is able to read the JSON data into their database to be used by the front end.

2.3. C.L.A.W.S. Scrapper

The C.L.A.W.S. scrapper is a scrapper that we developed in house to help us go beyond the limitations of the other tools we are using, such as limits and formatting issues. While most of our data still came from these other tools we believe that if further developed the C.L.A.W.S. scrapper would be invalluable to the continued development of this project.

2.3.1. Completed

- **Basic Scraping Capabilities:** This build does currently have the ability to effectively scrape data from target websites. At current, it is keyed to scraping Walmart, but can be easily keyed to target and scrape other websites.
- **Enhancement of Search and Navigation Capabilities for Scraping Tool:** Successfully developed and integrated advanced functions enabling custom search terms and efficient website navigation, specifically targeting major e-commerce platforms like Walmart, to streamline the data scraping process.
- **Implementation of Advanced "Waiting" Strategies for Scraping:** Executed the introduction of sophisticated wait strategies designed to minimize CAPTCHA activations, thereby enhancing the overall efficiency and reliability of web scraping activities.

2.3.2. Incomplete

- **Automonomy:** This scraper was not able to autonomously scrape websites without user intervention and oversight. This is the critical failure of the in-house CLAWS scraper. The amount of user intervention to solve CAPTCHAs, combined with the extreme ease of use of other enterprise scraping soultions (Blue Circle, Red Cart, etc), this scraper is not economically feasible for continued center stage use.
- **Deployement Environment Issues:** We were unable to resolve lack of compatability with some niche Selenium based requirements and our deployement environment. Specifically, we were faced with a single incompatability: ChromeDriver provided us with the needed stealth features to avoid the enterprise grade bot detection featured on Walmart and Texas State Bookstore (operated by Follet, a national 3rd party contractor for book store operations) for any reasonable amount of time. However, Linux Mint, the OS on the server, doesn't support the latest versions of the Chrome browser, which is required for the ChromeDriver to run. Other environments may provide more stablility and compatibility.

2.4. Coupon

The idea to implement a coupon tracker was to help users save more money and track the best deals. We recognized fairly early on that this would be something of a difficult addition due to the expiring nature of cupons and their exclusivity to particular stores.

2.4.1. Completed

The quick API is used to create the coupon. The coupon's JSON file is prepared, and the coupon code has been constructed. The creation and upload of the Coupon API code have been accomplished with success. In keeping with the project's original plan, which stressed the development of an approachable coupon application with a specific focus on Walmart and Best Buy products, this represents a noteworthy accomplishment. This phase's completion guarantees the application's fundamental functionality and prepares the way for future development and integration initiatives.

2.4.2. Incomplete

- The project's original goal was to smoothly include coupons for particular goods.
- It had a setback because coupons were not available for all of the products that were chosen.
- This detour from the initial goal required a change in emphasis.
- Rather than obtaining coupon codes for specific products, the new approach makes use of the coupon API to obtain coupon codes for entire websites and retailers.
- This shift in approach highlights the difficulties encountered during the coupon-sourcing process and the requirement to adjust for unforeseen circumstances.

QC Report

In the Quality Control (QC) section of our report for the Bobcat CLAWS project, we directed our efforts towards critical scripts that are integral to the system's functionality. Our team conducted exhaustive testing on the Parsehub.py script, which is designed to automate the scraping of product data from various online sources. We meticulously evaluated the script to confirm that it correctly retrieved environment variables, formatted URLs appropriately, and processed and de-duplicated JSON data effectively before database insertion.

We paid special attention to the script's ability to handle anomalies. For instance, we encountered a minor bug where product price data scraping from Best Buy's detail pages was inconsistent. While this did not disrupt the script's overall performance, it did necessitate manual intervention, pointing us towards potential enhancements in error handling. We also identified a low-severity bug related to the failure in scraping category and subcategory data from the same pages, which was mitigated by our existing error-handling procedures.

For the tAPI.py and wmAPI.py scripts, which are responsible for interfacing with the Target and Walmart APIs respectively, we found no bugs, suggesting their stability and reliability in the current build.

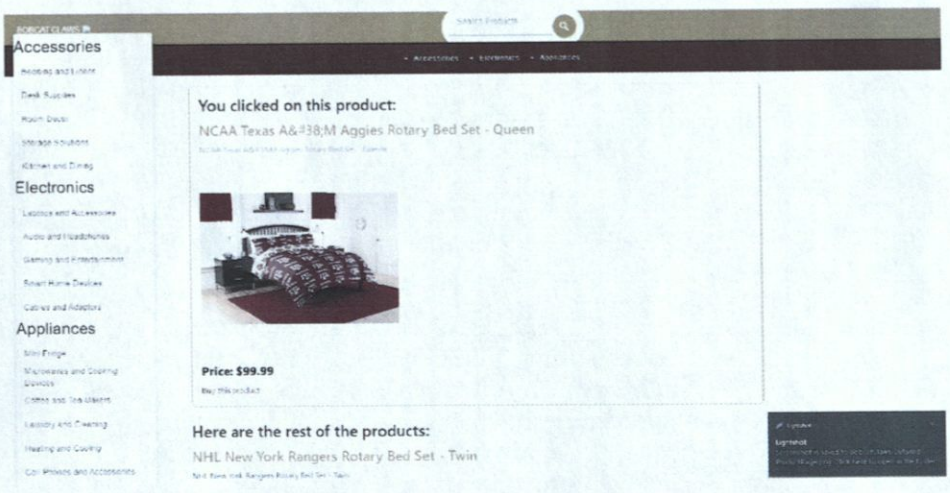
Our QC findings indicate that while the level of functional testing was satisfactory, there is a clear avenue for us to extend our test cases to cover a broader range of functions in finer detail. By doing so, we can ensure a more rigorous validation of our system's robustness and a higher degree of confidence in its deployment readiness. These insights from our QC process underscore our commitment to ongoing enhancement of our testing strategies to bolster system dependability.

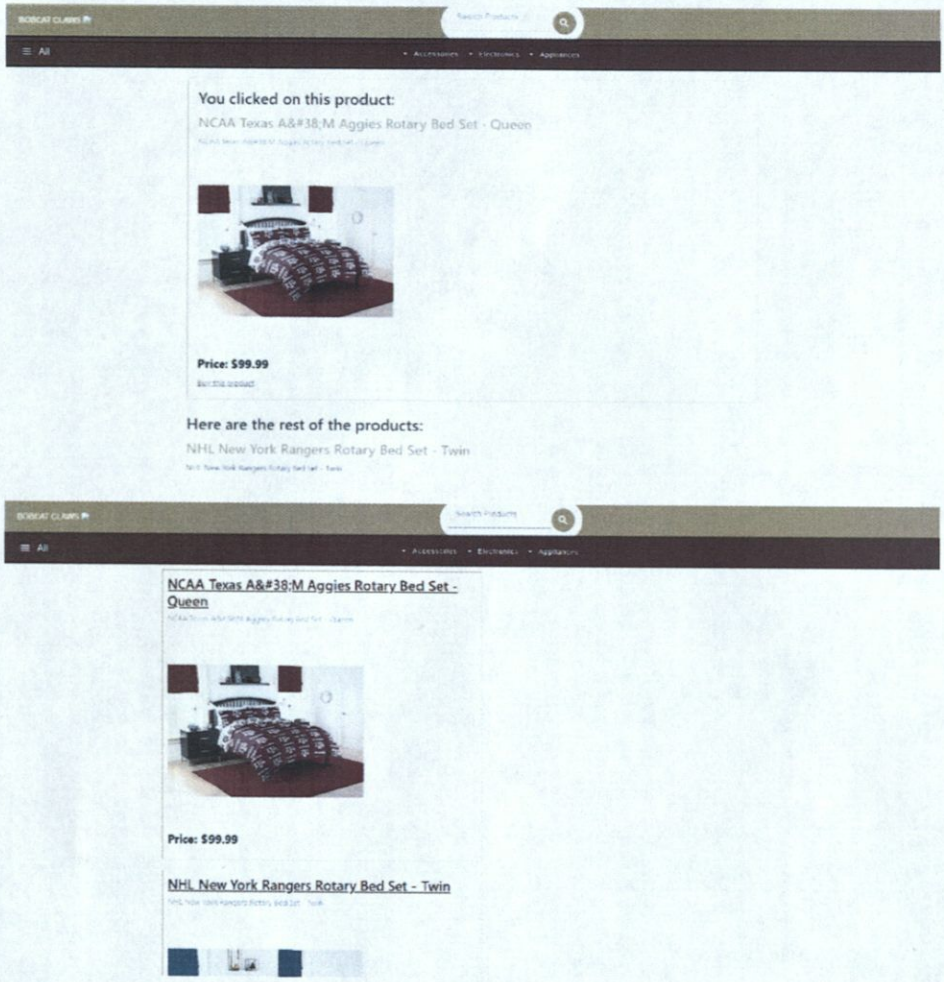
Supporting Images

Front-end

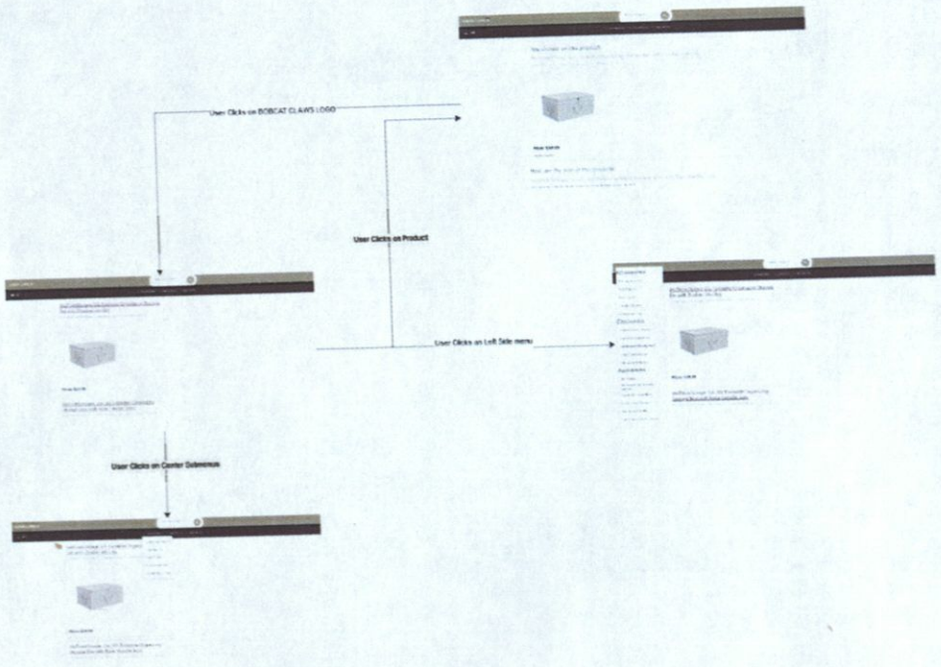
Logo

College Lifestyle Affordable Wares Search

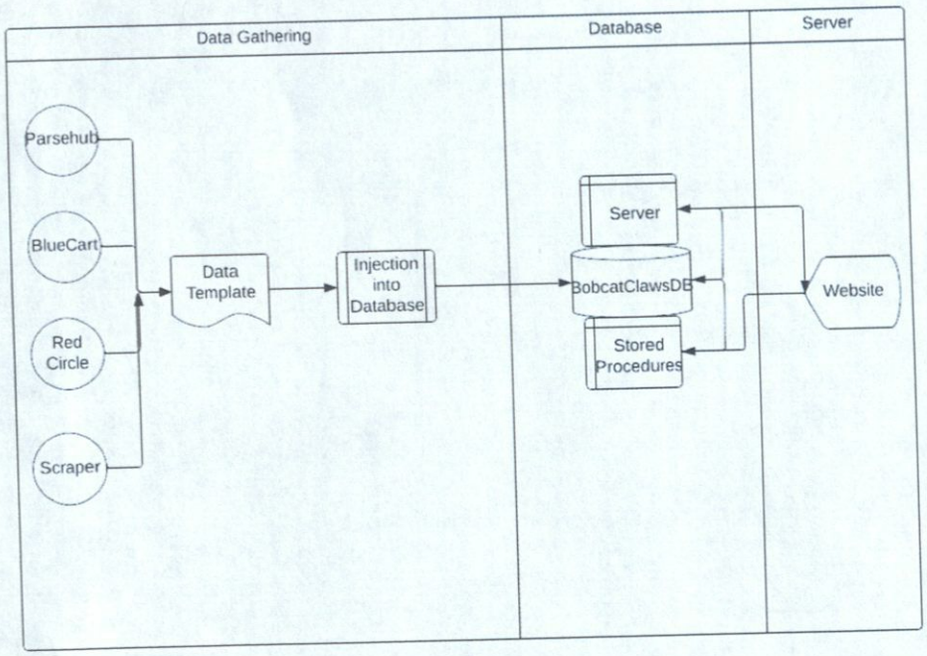




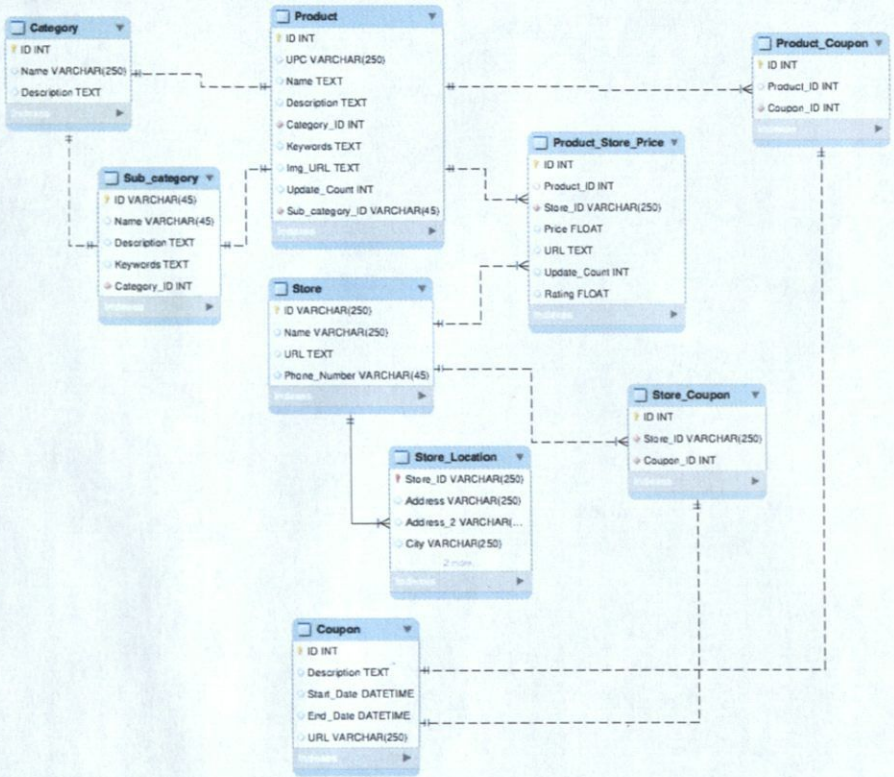
Customer Flow Diagram

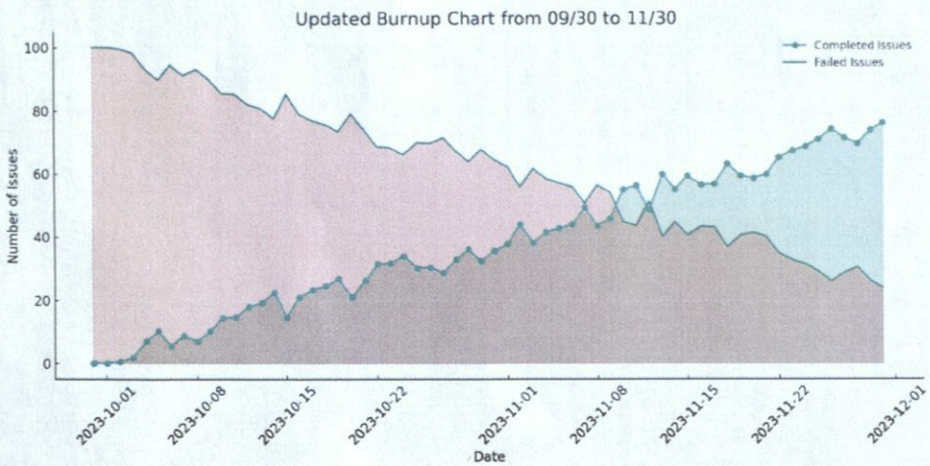


Dataflow for BobcatClaws Product Comparison Website



Database Design Diagram


Category
ID INT
Name VARCHAR(250)
Description TEXT
Product
ID INT
UPC VARCHAR(250)
Name TEXT
Description TEXT
Category_ID INT
Keywords TEXT
Img_URL TEXT
Update_Count INT
Sub_category_ID VARCHAR(45)
Product_Coupon
ID INT
Product_ID INT
Coupon_ID INT
Product_Store_Price
ID INT
Product_ID INT
Store_ID VARCHAR(250)
Price FLOAT
URL TEXT
Update_Count INT
Rating FLOAT
Sub_category
ID VARCHAR(45)
Name VARCHAR(45)
Description TEXT
Keywords TEXT
Category_ID INT
Store
ID VARCHAR(250)
Name VARCHAR(250)
URL TEXT
Phone_Number VARCHAR(45)
Store_Coupon
ID INT
Store_ID VARCHAR(250)
Coupon_ID INT
Store_Location
Store_ID VARCHAR(250)
Address VARCHAR(250)
Address_2 VARCHAR(...
City VARCHAR(250)
Coupon
ID INT
Description TEXT
Start_Date DATETIME
End_Date DATETIME
URL VARCHAR(250)


Updated Burnup Chart from 09/30 to 11/30
Completed Issues
Failed Issues
Number of Issues
0
20
40
60
80
100
2023-10-01
2023-10-08
2023-10-15
2023-10-22
2023-11-01
2023-11-08
2023-11-15
2023-11-22
2023-12-01
Date