DATE: April 22nd, 2019

TO: Dr. Mary Eberlein, TA's

FROM: Team Eagle - Hiep Nguyen, Michael Flanders, Kunpeng Qin, Nabil Khan, Nelson Levy, Wesley Klock

SUBJECT: Team Eagles' BookBrain Phase 3 Report

We are team Eagles on Canvas, and our project is BookBrain. Our github can be found here: https://github.com/Flandini/461L-Eagles, and our webapp is located at http://bookbrain.club/. We have 6 members on our team:

- Name: Hiep Nguyen, Email**:** hiepnguyen8069@gmail.com, github username: hnguyen257
- Name: Michael Flanders, Email**:** flanders.michaelk@gmail.com, github username: flandini
- Name: Nelson Levy, Email**:** nlevy2807@gmail.com, github username: nrl538
- Name: Wesley Klock, Email**:** wesleyklock@gmail.com, github username: wesclock777
- Name: Kunpeng Qin (Phase 3 Leader), Email**:** qinkunpeng2015@utexas.edu, github username: qkpqkp
- Name: Nabil Khan, Email**:** nabilliban14@gmail.com, github username: Nabilliban14

TASKS, FEATURES, AND FUNCTIONALITY

For the third phase, we promised five things:

- Dynamic web pages
- Dynamic web page tests
- Improved book searching and filtering
- More sources and thousands of books listed
- A fully functioning web application

For this phase, we focused on finishing our scraper by adding all scraper sources, making our web pages dynamic, scraping more book data, continuing to write tests, and displaying sentiment analysis of book reviews.

For our user interface (UI), we improved the overall mobility of the frontend and information presentation. We added additional UI elements that change without page reload such as the “Save” and “Remove book from favorites” buttons, our dynamic critic review loading for books, and the similar books carousel. Additionally new content regarding sentiment of reviews and an explanation of these metrics are displayed.

We have increased our number of tests from 32 to 58 and have unit tests, integration tests, and dynamic web content tests as promised for this third phase. We decided to implement dynamic testing using Selenium to make sure our front-end was reacting as intended. These tests were implemented in Java using JUnit and ran via Maven.

We have added Amazon, Twitter, and Reddit to our previous scraper sources: Barnes & Noble, IDreamBooks, and GoodReads. We have also added the IDreamBooks widget to each book’s page that have critic reviews available. We are now also displaying a “Polarity” and “Subjectivity” metric for each type of review on each book’s page.

We have implemented extra features such as a user profile page where users can view their recently viewed books and saved/favorited books.

DESIGN

We are using Python2.7 and Flask for our web application. Because there are no classes in our scraper, testing, or web app code, we have not included a UML class diagram as we felt that it would not properly describe the design of our application. Instead, we have included a UML sequence diagram below to describe the design of our web application for this third phase. The diagram was rather large, so you may need to zoom in to properly see the text.

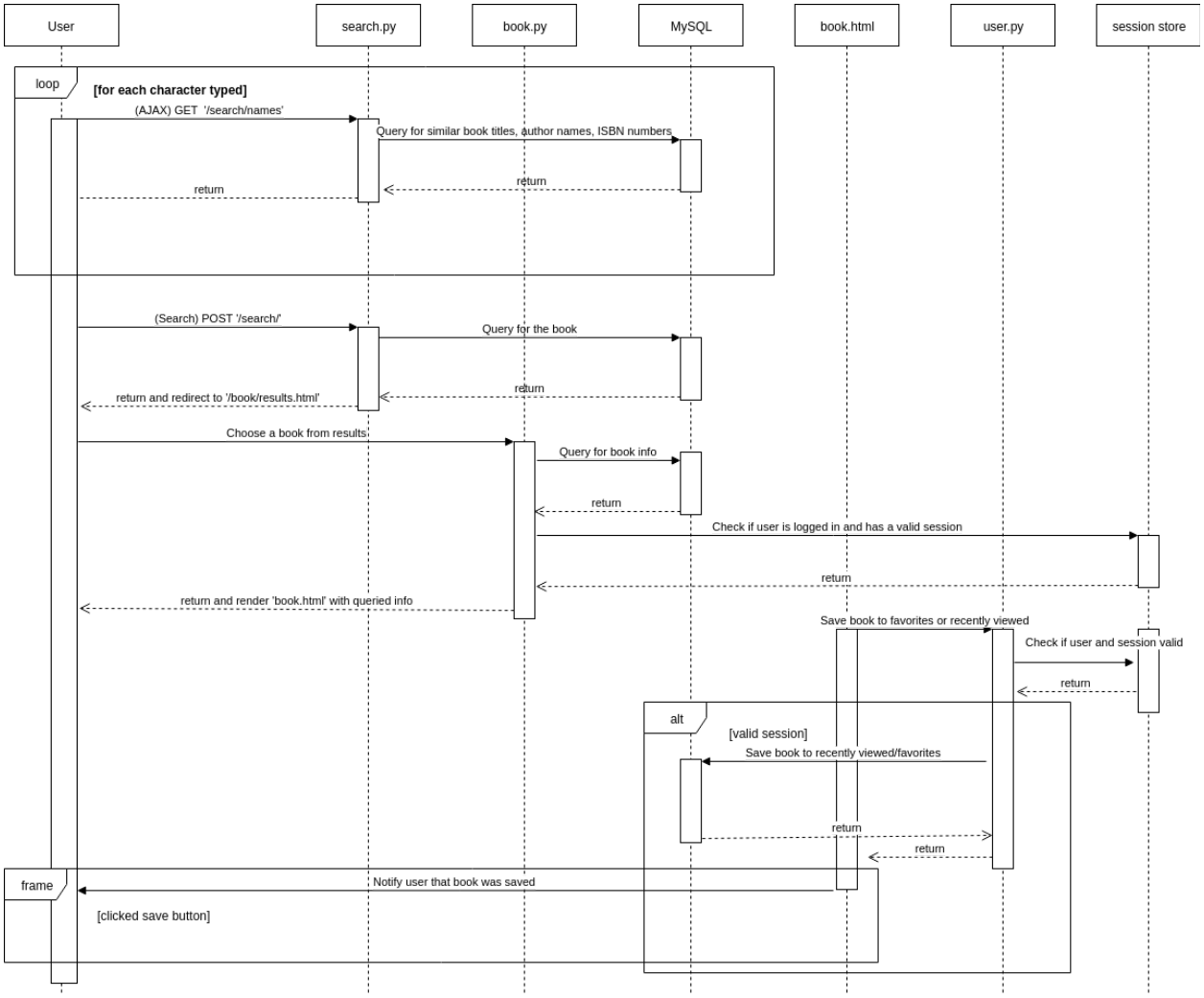

Figure 1. Sequence Diagram depicting autocomplete, search, and book page display with dynamic save to favorites/recently viewed

Python 2.7 was used for scraping code. For scraping we used APIs where possible to simplify and speed up the gathering of information. If no API was available, Selenium was used to navigate the site and gather the information using BeautifulSoup to parse the site's source. The scrapers use an input of the main csv file we are using for all our books and output a csv with key information from the main csv as well as the added information that was gathered.

REQUIREMENTS

We introduced 5 new user stories for this phase that are available on our Github issues board (https://github.com/Flandini/461L-Eagles/issues/4):

- As a person looking for trendy books and reviews, I want to be shown reviews and sentiment of a book from social media sites.
- As a person looking for a new book to read, I want to be shown books and their covers similar to the current one I'm looking at.
- As a person who isn't great at remembering every book I've looked at, I would like to be able to register, login, and save my favorite books and their reviews.
- As an analytical person, I would like to know how subjective listed book reviews are.
- As an analytical person, I would like to know the sentiment and emotion of listed book reviews.

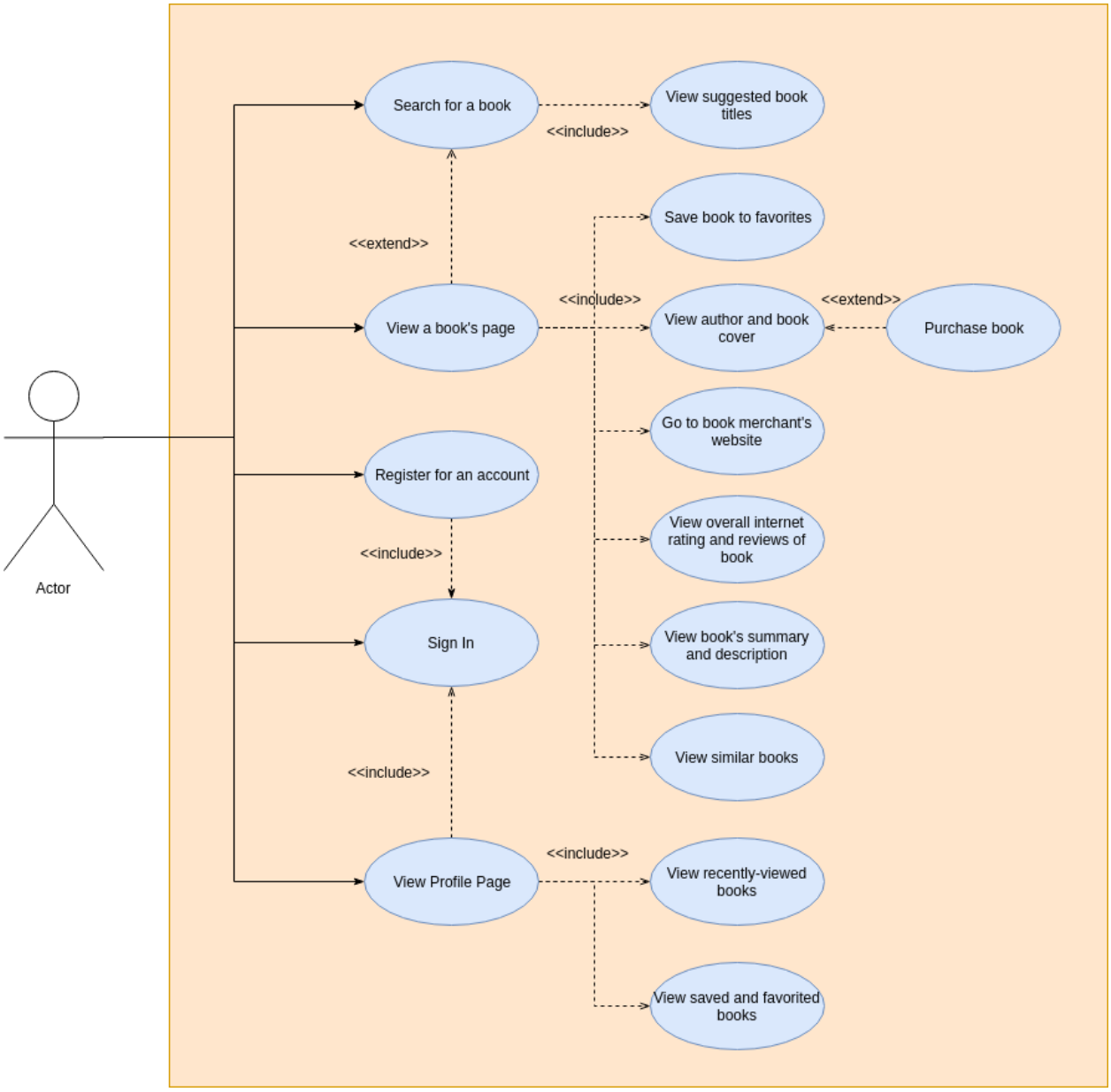

Figure 2. Phase 3 Updated Use Case Diagram

This phase's user stories focused on users being able to make accounts and save books while presenting more data per book to the user. We have implemented and fulfilled all of the requirements outlined by these user stories and have shown this in Figure 2 below. There are differences between Figure 3, our goal use case diagram for our finished web application, and what we have actually implemented shown in Figure 2. The reason for these differences is that

during our Agile development process, we decided to take a more user-oriented approach to the website. As a result, we did not end up fulfilling the use cases of users being shown a list of trendy books on the homepage or the use case of being able to filter by genre. Instead, we have fulfilled new use cases of being able to register for an account, login to that account, save books to a favorites list, and view recently viewed books. We have also implemented functionality to show users similar books to the ones they have viewed, but there is no genre support.

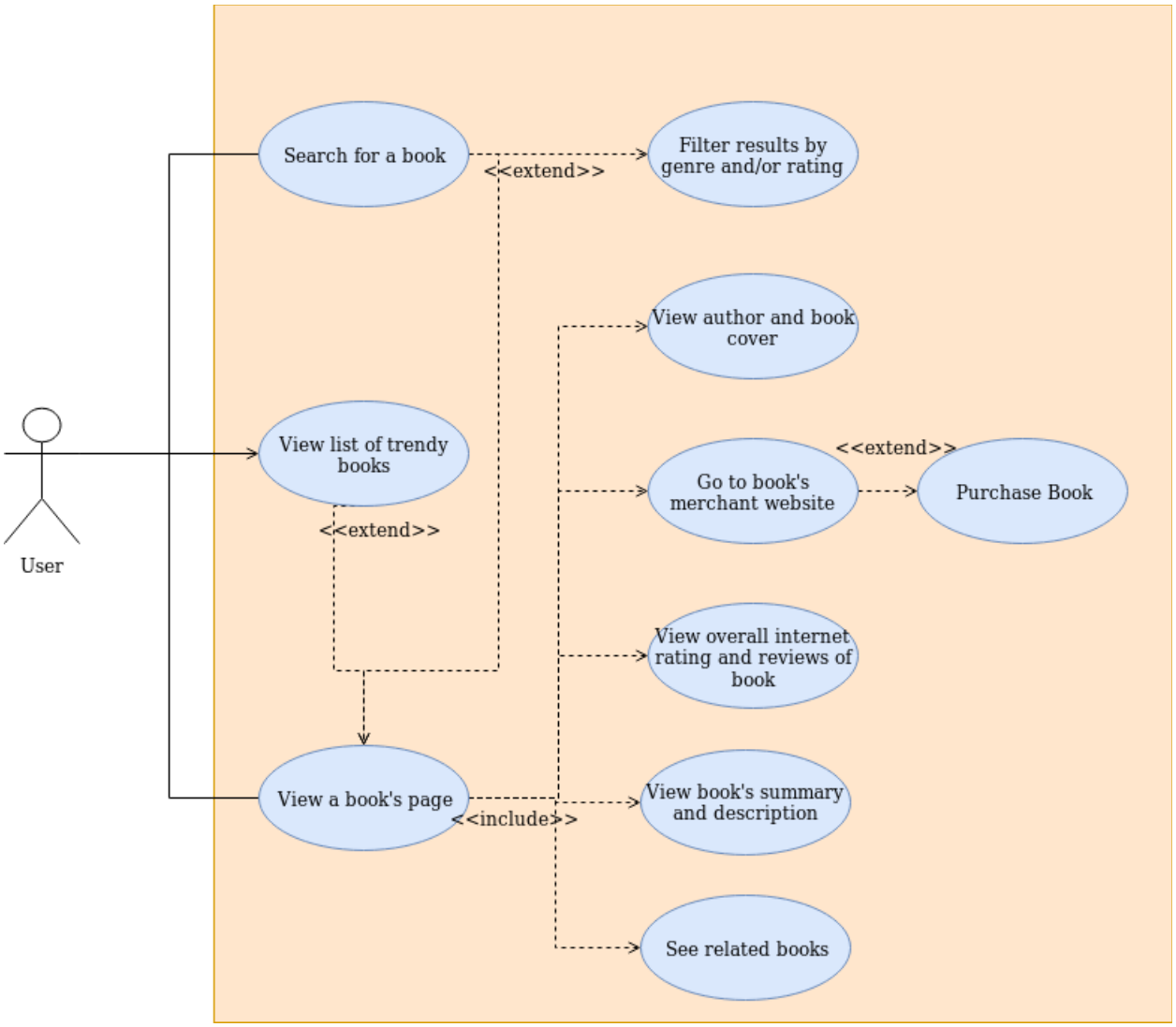

Figure 3. Goal Use Case Diagram/Use Case Diagram of BookBrain at Completion per Proposal

Screenshots of our finished web application, BookBrain:

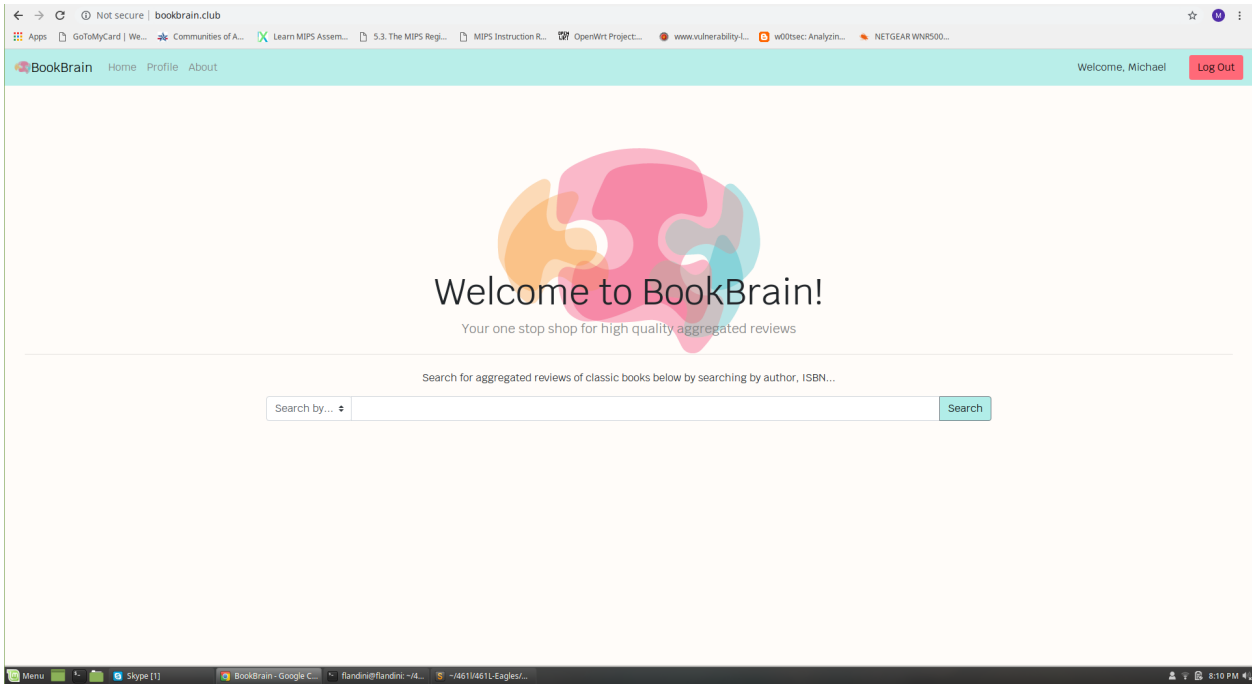

Figure 4. BookBrain Home Page with User Logged In

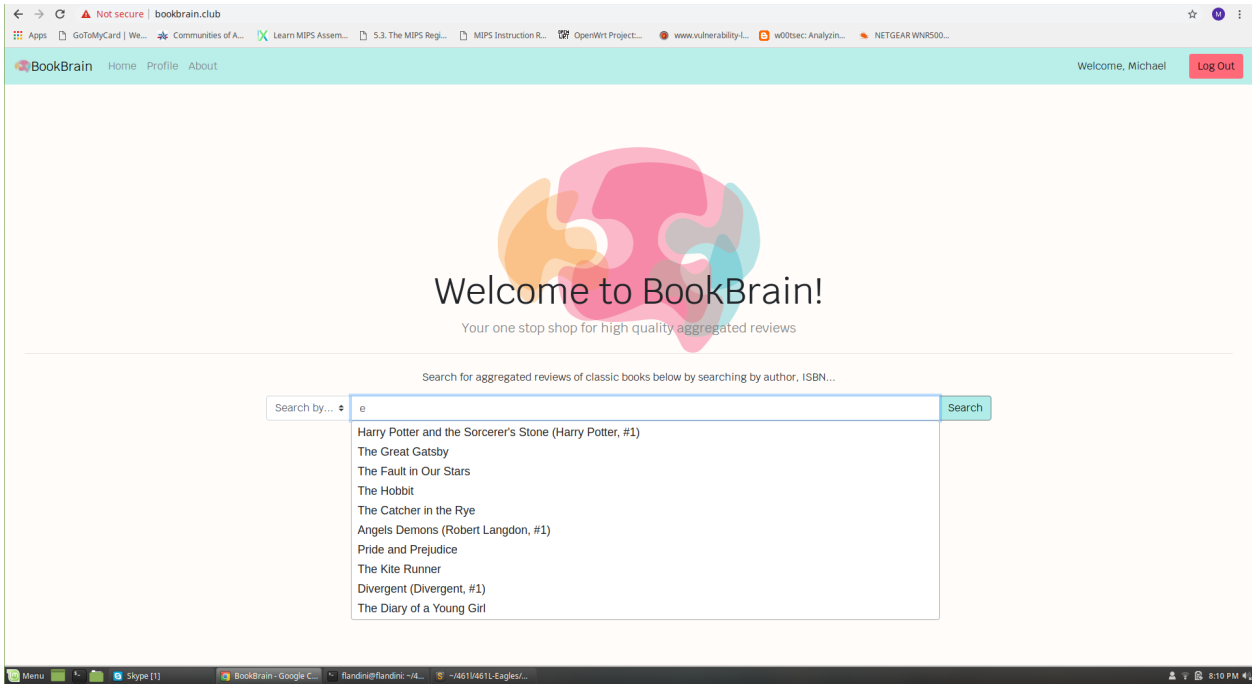

Figure 5. BookBrain Autocomplete Functionality

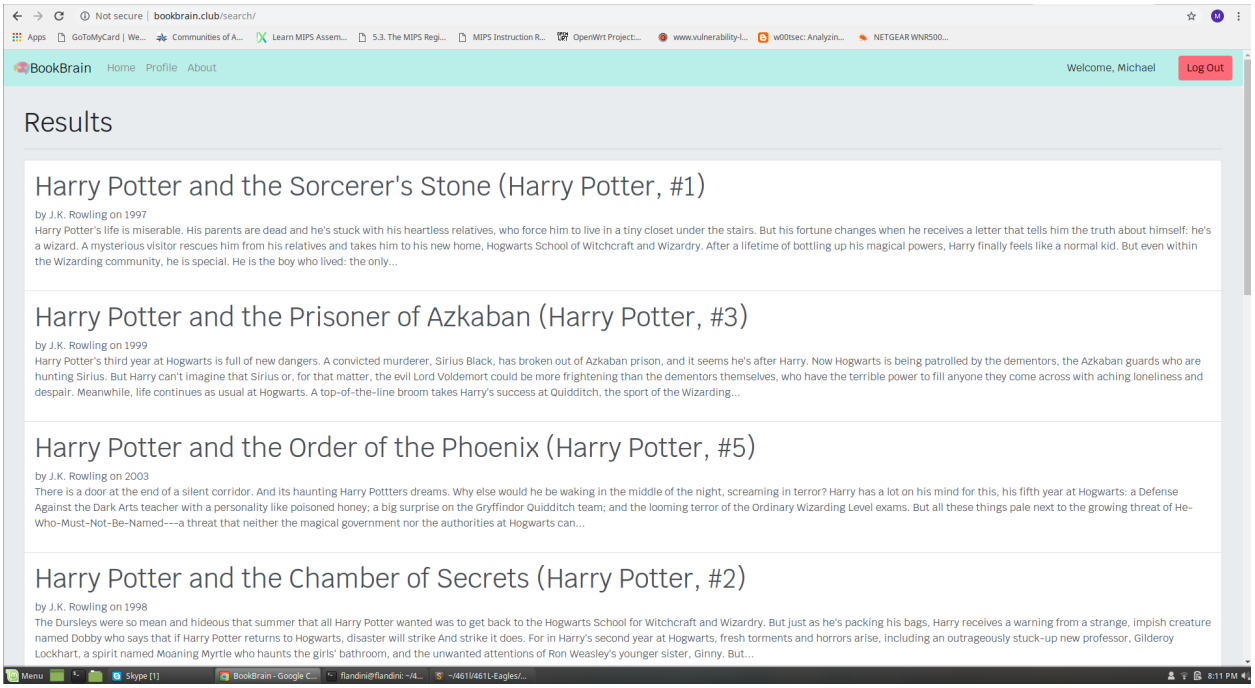

Figure 6. BookBrain Search Results Page

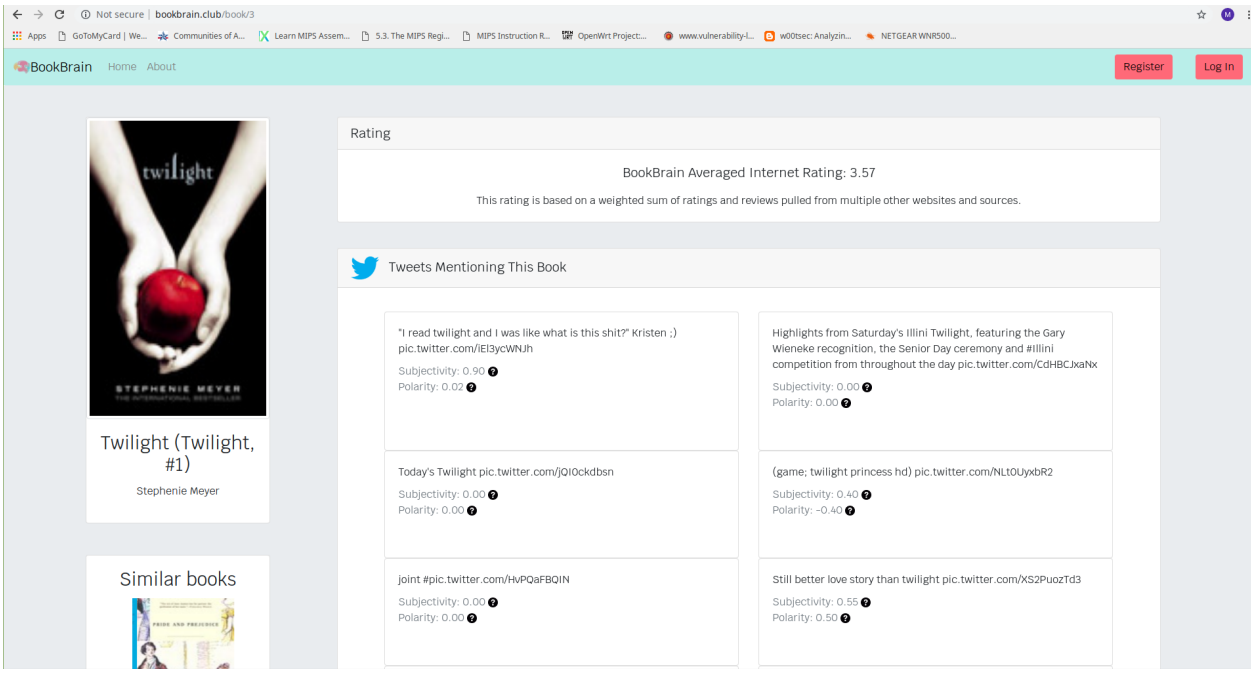

Figure 7. Screenshot 1 of 4 of Individual Book Page (No User Logged In)

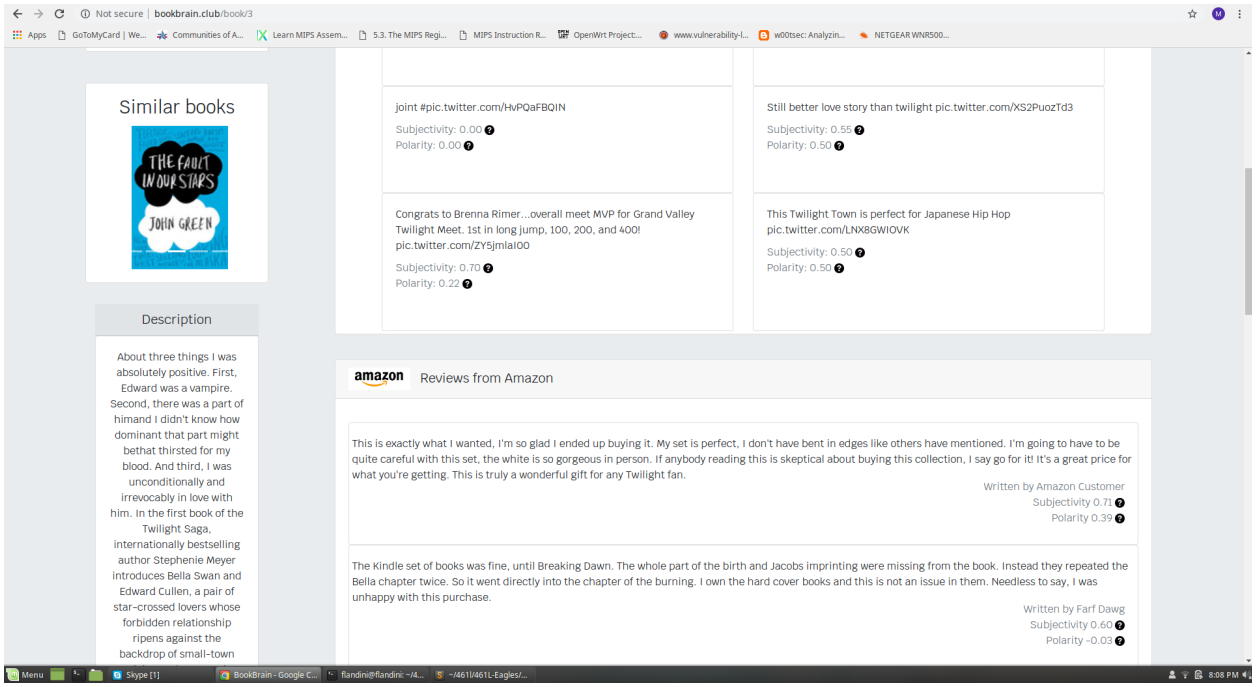

Figure 8. Screenshot 2 of 4 of Individual Book Page (No User Logged In)

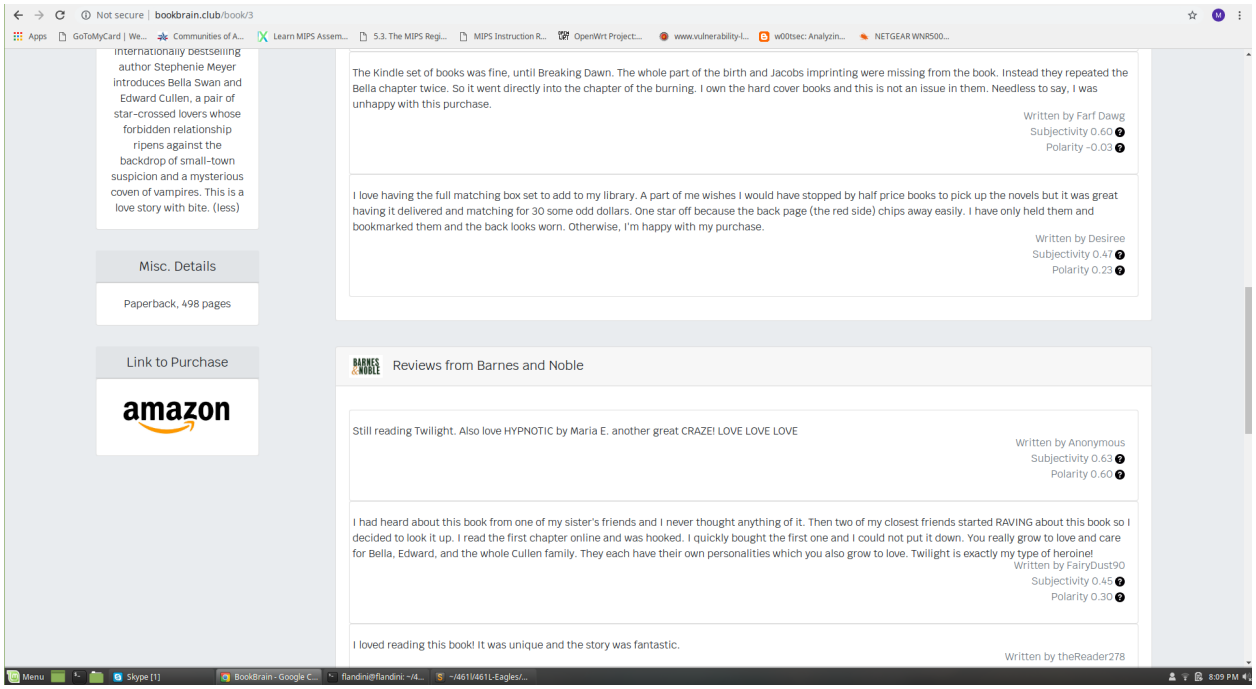

Figure 9. Screenshot 3 of 4 of Individual Book Page (No User Logged In)

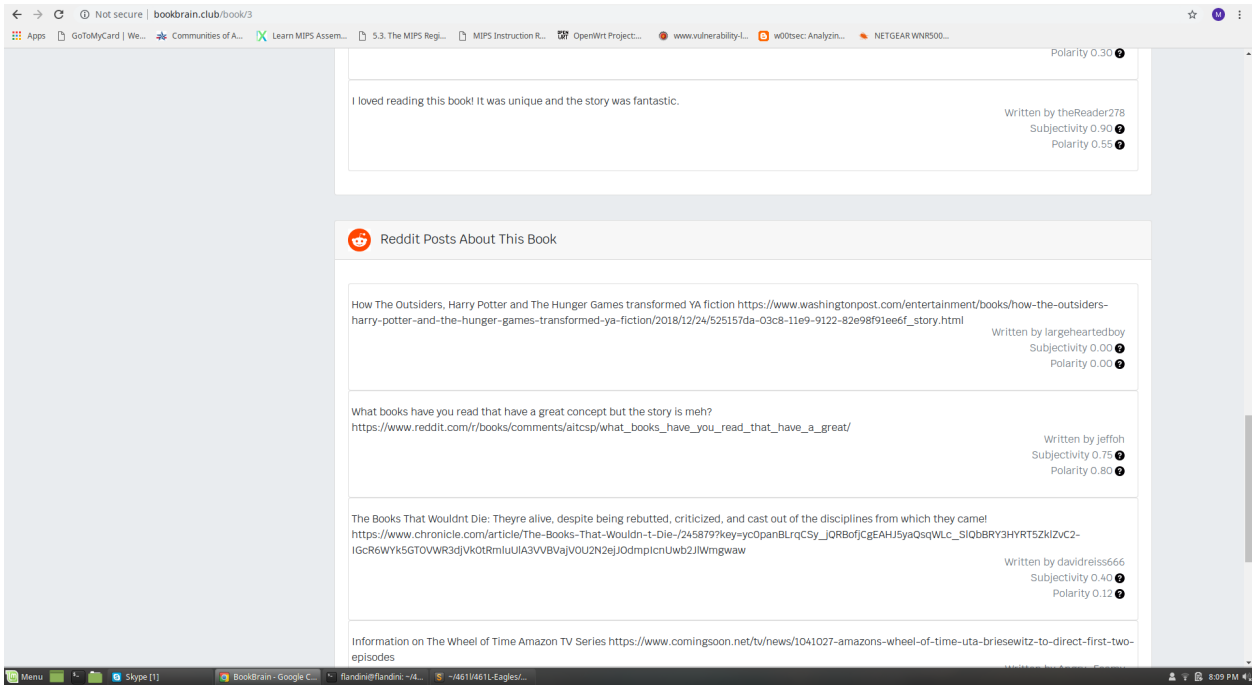

Figure 10. Screenshot 4 of 4 of Individual Book Page (No User Logged In)

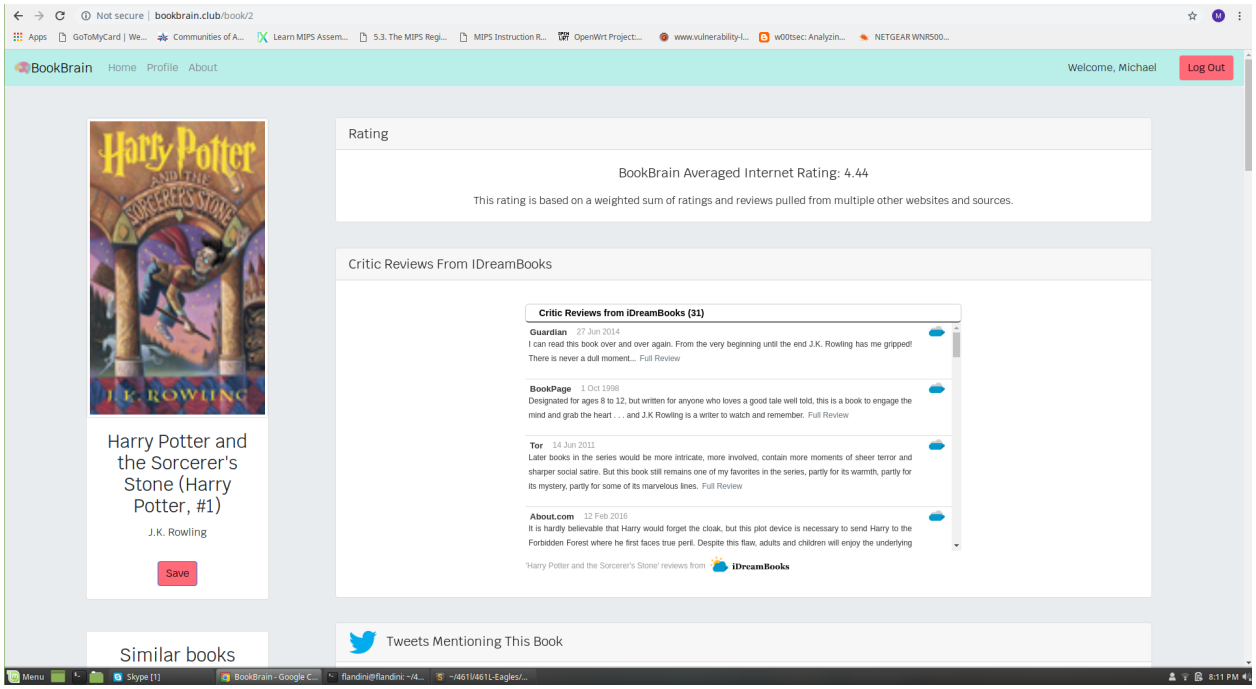

Figure 11. Screenshot of Individual Book Page with User Logged In and Save Button Below Book Title

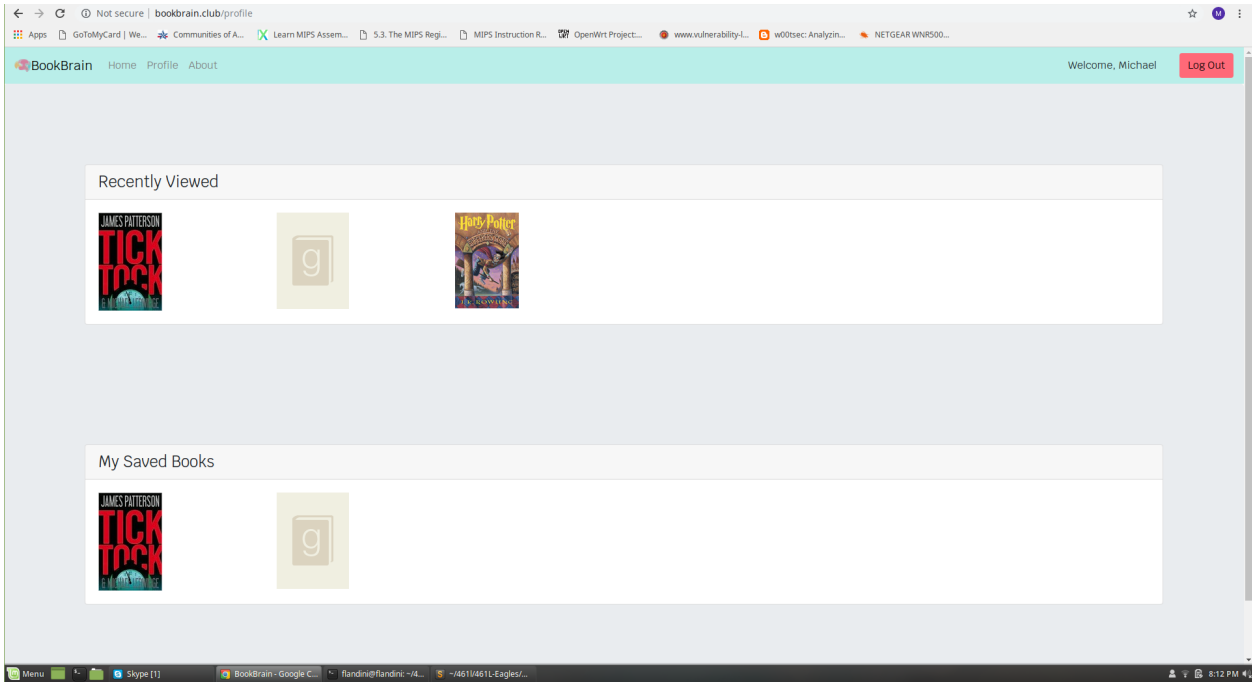

Figure 12. Screenshot of User Profile Page Showing Three Recently Viewed Books and Two Saved Books

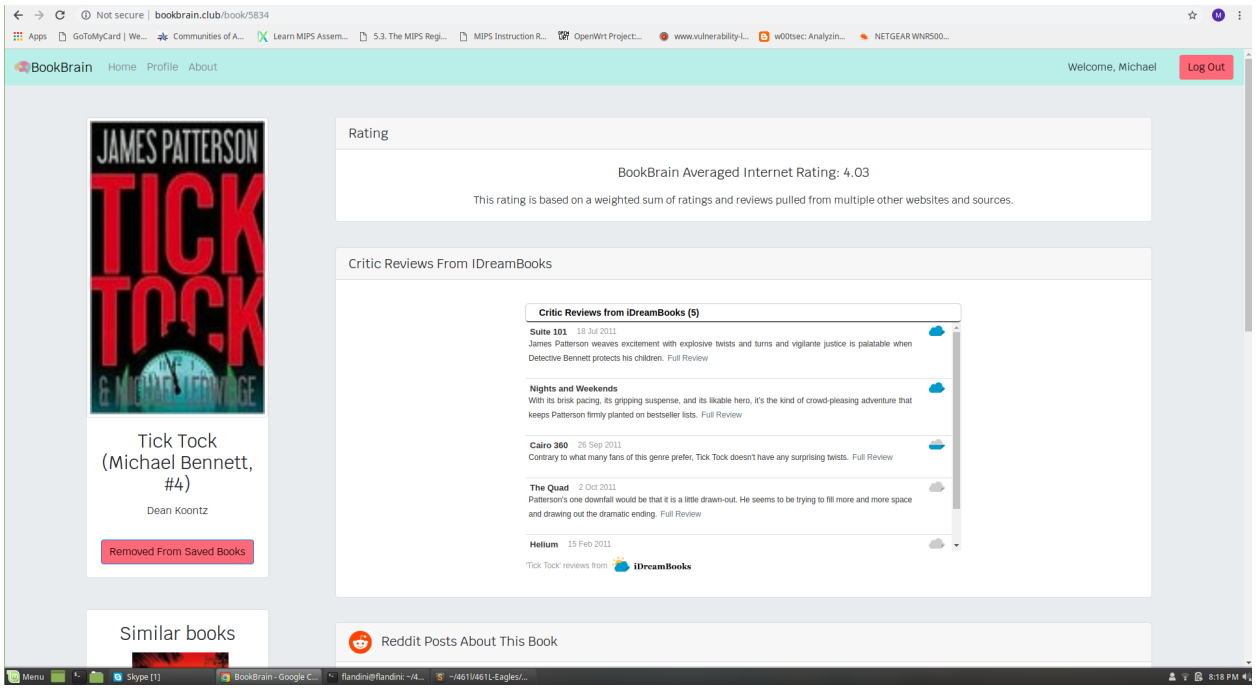

Figure 13. Screenshot of Individual Book Page with ‘Remove From Saved Books’ Button and Critic Review Widget

TOOLS, SOFTWARE, AND FRAMEWORKS

In addition to all of the tools, software, and frameworks used in previous phases, we have added only a few more tools to our web application. In this phase, we have used Maven and Selenium for testing. We have also added sentiment to each review by using TextBlob. To manage our project, we have created an issue board on Trello. We have used Google App Engine to hold our web app and Google Cloud SQL to hold our Mysql database.

TESTING

In the previous phases we had only incorporated static tests, but in this phase we utilized Selenium, a framework tool for testing websites, to incorporate dynamic tests. More specifically, we utilized Selenium WebDriver to load our web application and interact with its web elements. Figure 14 shows the project file structure and the code for creating and destroying a web driver. To organize our tests, we created a test java file for each major web page on our website: HomepageTesting.java, ResultsPageTesting.java, and BookPageTesting.java, AboutPageTesting.java. In the following paragraphs, we will describe each test file in detail.

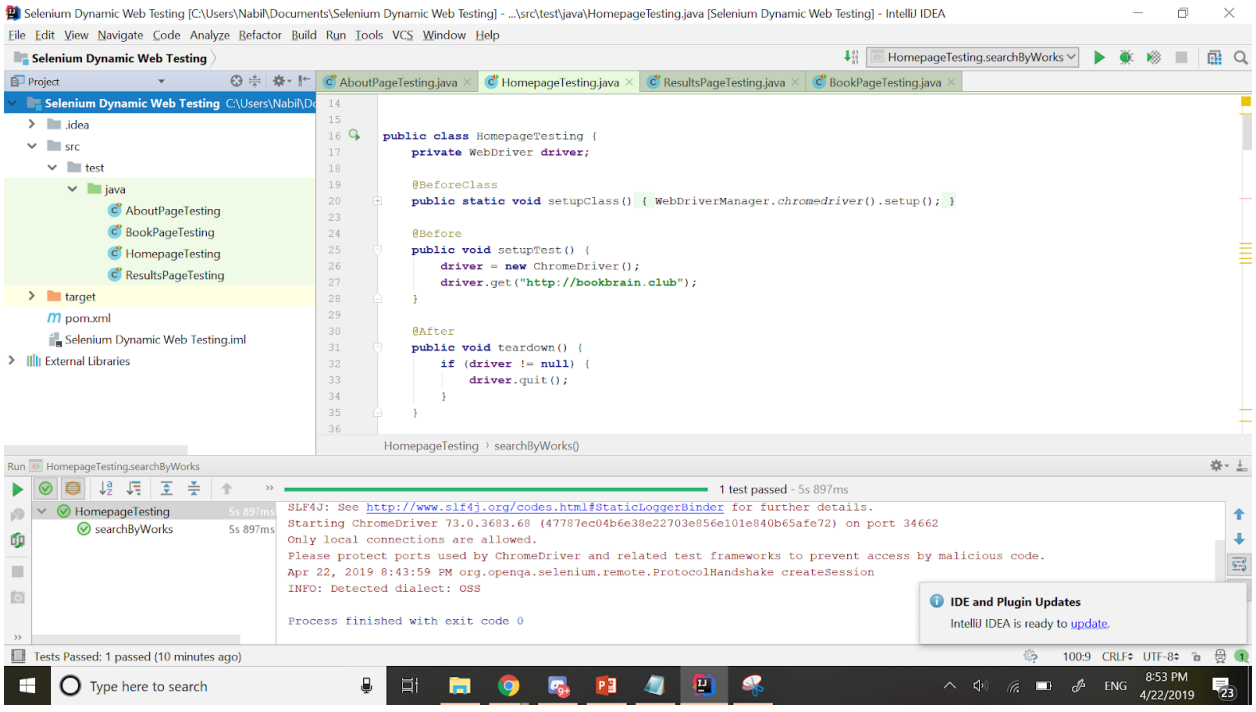

Figure 14. Test Project Structure and WebDriver Code

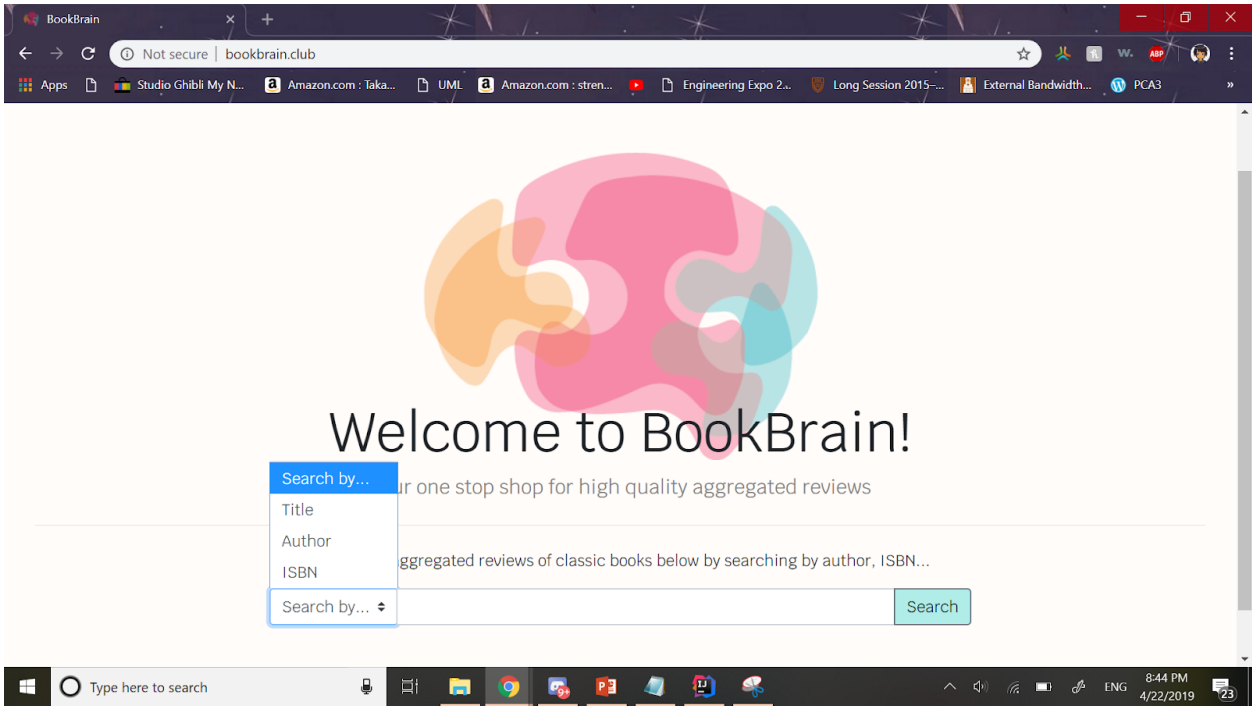

Figure 15(a). Search By Element Test

```
@Test
public void searchByWorks() {
    WebElement searchOptions = driver.findElement(By.id("searchOptions"));
    Select searchSelect = new Select(searchOptions);

    assertEquals( expected: "Search by...", searchSelect.getFirstSelectedOption().getText().toString());

    searchSelect.selectByVisibleText("Title");
    assertEquals( expected: "Title", searchSelect.getFirstSelectedOption().getText().toString());

    searchSelect.selectByVisibleText("Author");
    assertEquals( expected: "Author", searchSelect.getFirstSelectedOption().getText().toString());

    searchSelect.selectByVisibleText("ISBN");
    assertEquals( expected: "ISBN", searchSelect.getFirstSelectedOption().getText().toString());

    searchSelect.selectByVisibleText("Search by...");
    assertEquals( expected: "Search by...", searchSelect.getFirstSelectedOption().getText().toString());
}
```

Figure 15(b). Search By Element Test Code

In HomepageTesting.java, we made tests to ensure that every clickable web element redirects the user to the correct link. Most of our detailed testing for this webpage went into the search form. The search options were tested for responsiveness when requested to change, seen in Figure 15(a) and Figure 15(b). The autocomplete function of the form was tested for correctness, such

as ensuring it did not offer title suggestions when a user typed in numbers with the ISBN search by option selected.. It was also tested to ensure it offered at most 10 suggestions, because we wouldn't want users to be given an excessively long list.

In ResultsPageTesting.java, we mainly focused on three things. Results for a search are displayed, results are clickable, and the pagination buttons on the bottom of the page are functioning as they should. For web elements in general, we need to find the web element tag that will allow Selenium's webdriver to interact with it. This is usually done by using the inspect option on Google Chrome, and then applying the ID given in our code, as seen in Figure 16(a) and Figure 16(b).

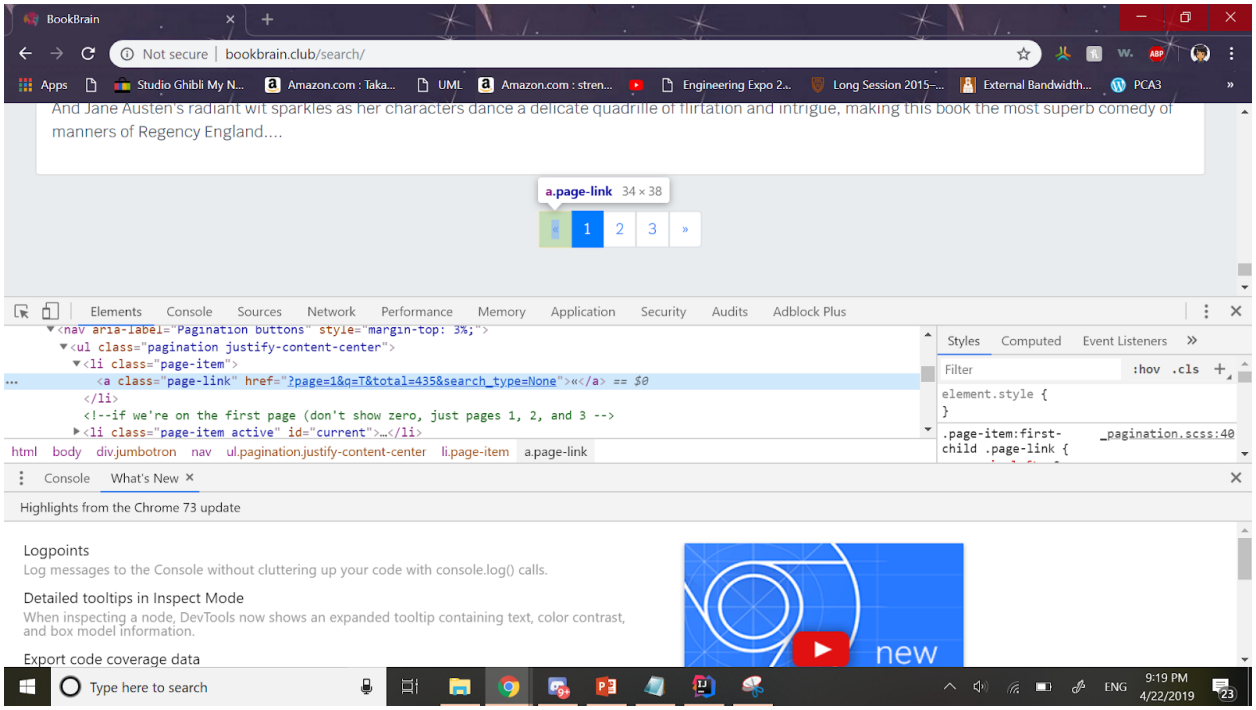

Figure 16(a). First Page Pagination Test and ID

```
@Test
public void paginatorClickingFirstPageWorks() {
    WebElement paginator = driver.findElement(By.cssSelector("ul.pagination.justify-content-center"));
    WebElement lastPageButton = paginator.findElements(By.tagName("li")).get(0);

    //make sure button I am clicking is the << button
    assertEquals( expected: "«", lastPageButton.findElement(By.tagName("a")).getText().toString());
    lastPageButton.click();
    driver.manage().timeouts().implicitlyWait( l: 3, TimeUnit.SECONDS);

    paginator = driver.findElement(By.cssSelector("ul.pagination.justify-content-center"));
    String pageItShouldBe = paginator.findElements(By.tagName("li")).get(1).getText().toString();
    String currentPage = driver.findElement(By.cssSelector("li#current.page-item.active")).getText().toString();

    assertEquals(pageItShouldBe, currentPage);
}
```

Figure 16(b). First Page Pagination Test and ID Implementation

In BookPageTesting.java, we made tests to ensure that links and buttons worked just like the previous files. In this case, it was specifically the buy button and similar books icon as seen in Figure 8 and Figure 9 respectively. On top of that, however, we created further tests to ensure that the iDreamBooks widget (seen in Figure 13) was working and interactive.

Finally, in AboutPageTesting.java, we made sure the hyperlinks listed would link to the correct website. This was simple as we simply had to compare the Selenium webdriver's current URL after clicking the hyperlink with what we expect it to be, and using AssertTrue to ensure the test passes the way we want to.

By breaking the web application into its four major web components and testing the dynamic elements in each of those major web components, we were able to do a full test coverage of the different things the user could interact with. In total, we performed 26 different Selenium Dynamic Webpage tests on BookBrain. Below are the figures representing the passed test cases.

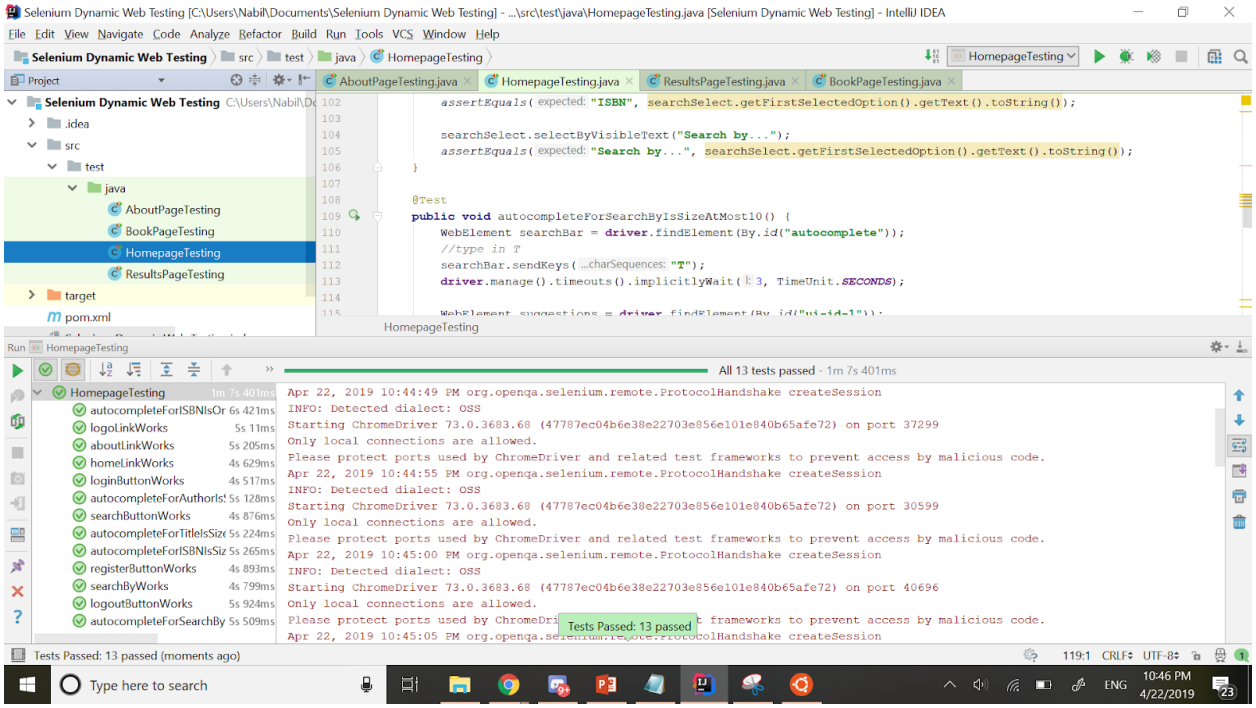

Figure 17. HomepageTesting.java 13 Test Cases All Passed

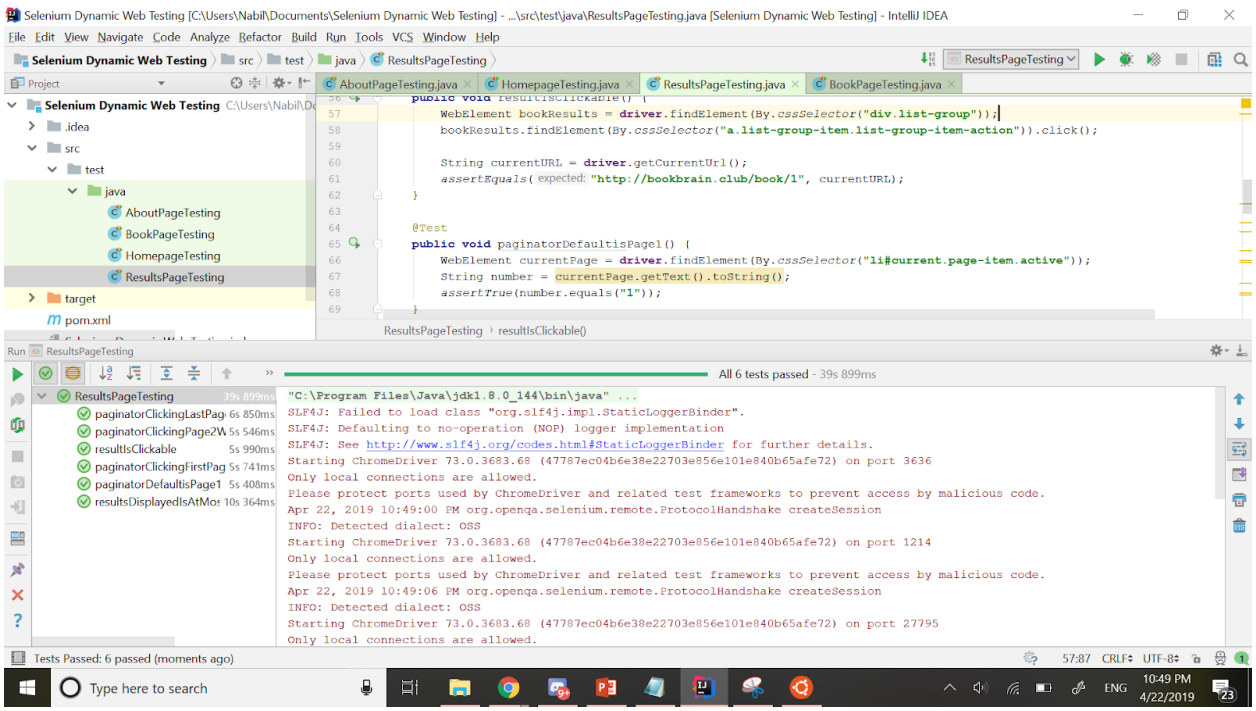

Figure 17. ResultsPageTesting.java 6 Test Cases All Passed

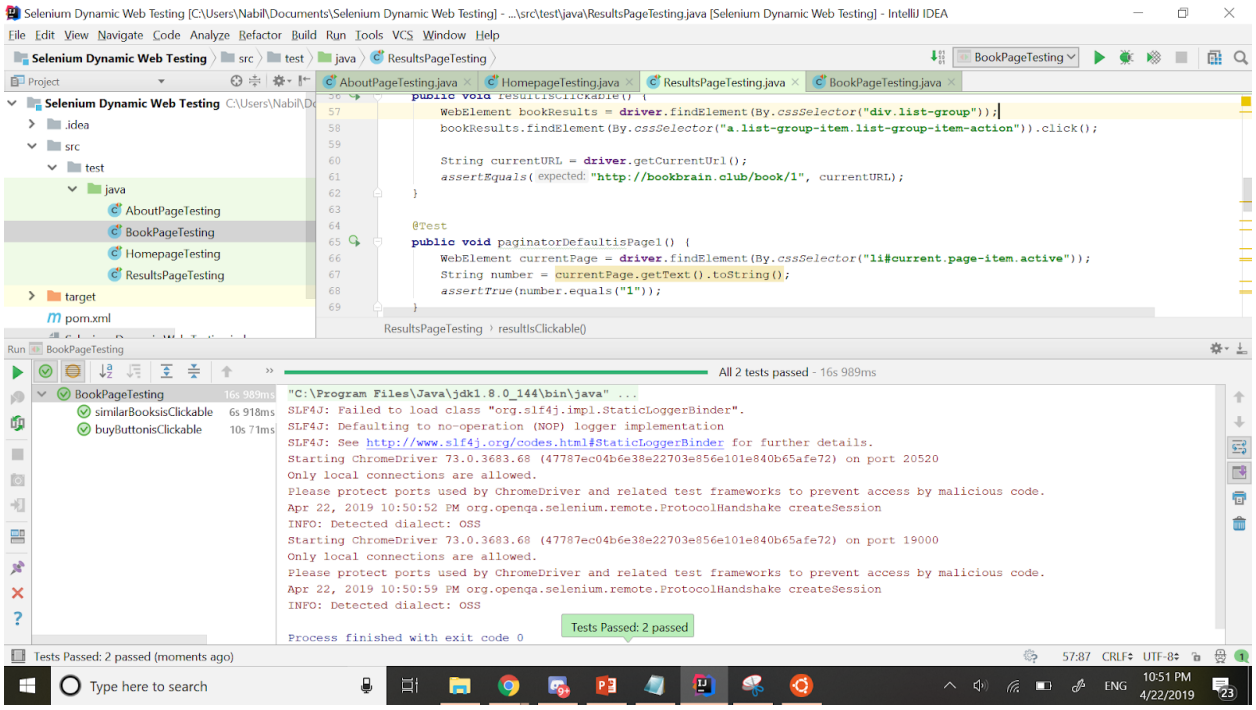

Figure 17. ResultsPageTesting.java 2 Test Cases All Passed

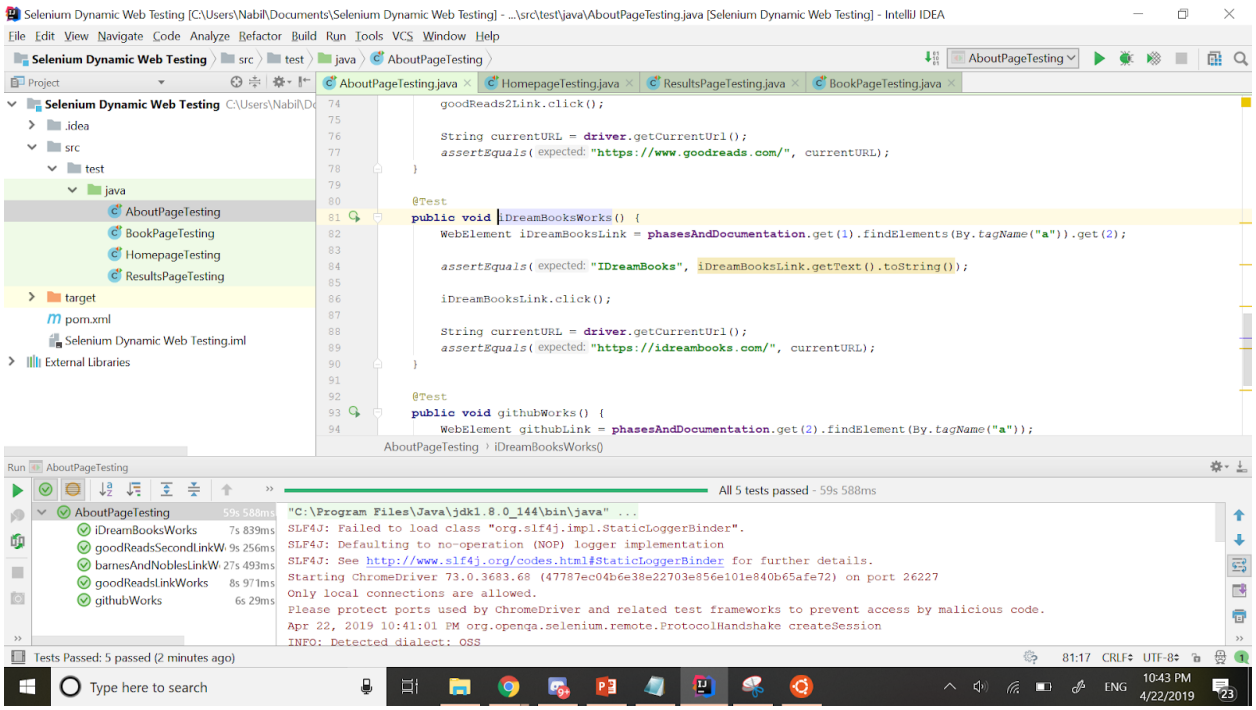

Figure 17. AboutPageTesting.java 5 Test Cases All Passed

FINAL REFLECTIONS

The biggest difficulties we faced with launching BookBrain were communication, infrastructure, collaboration, and large scrape times. Initially, we had problems with team communications such as setting due dates and meeting times, and there were also problems with collaboration on GroupMe. After meeting with Dr. Eberlein, we fixed these issues by using a Trello board to keep track of all issues and deadlines and by setting more in person group meetings.

Another collaboration and infrastructure issue that we ran into was our Github repository usage. Except for the end of the third phase, we all checked code into the repository by directly pushing commits to the main master branch. Sometimes we would end up overwriting or deleting others' commits whether by force pushing or just honest git mistakes. However, we realize that we should have forked the repo and used pull requests; this is something that we would do differently starting over.

We also faced difficulties in keeping data compatible and correct between different formats and people. For instance, it was difficult synching up everyone's databases with the current database schema in the Github repository. Even after getting everyone's databases on the same page, it was tough loading data from from CSV files into MySQL from scrapers that came from multiple people using different encodings and formats. If we could start development over, we would probably use something like Docker or Vagrant to enforce the similarity of development environments across team members and also define a specific format for data before distributing work.

Another one of our largest issues was dealing with large scrape times and incomplete scrape data. For this project, we had to scrape a lot of data from several different sites. We started out with scraping data from GoodReads and then using this scraped data to help scrape other sites. For example, after pulling book ISBN's from GoodReads, we used these ISBN's for our critic review widget and to reference which books we needed to scrape from other sites. However, it took a while to scrape data from any of the sites we specified, and if the scraped data from one of

the sites (especially GoodReads) was messed up, it meant having to re-scrape that site and others due to coupling of our data. Because of this coupling and the amount of sites that had to be scraped with each addition of new books, we did not get as many books scraped as we had hoped to.