David Fan (Leader) Roland Fong Nathanael Ji Kyle Xiao

TigerTexts

Final Report

COS 333 Spring 2018

I. Planning

A. Problem Statement

TigerTexts addresses the problem of how students can efficiently get coursebooks at the best possible prices. When buying coursebooks, students have to simultaneously consider online providers like Amazon, Labyrinth, and student hand-me-downs, in order to make a well-informed decision. Gaining perfect information about the textbook market for one's courses is a tedious process, and one that can be made more efficient with a web app. **TigerTexts** streamlines the process by displaying required textbooks for each course with prices and purchase links, as well as an in-app exchange platform that marks when books are sold - a replacement for the Facebook Textbook exchange - all in one interface.

Our goal was to integrate book information from multiple heterogeneous sources: Amazon, Labyrinth, and the Facebook Textbook Exchange. Due to the Facebook-Cambridge privacy scandal in March 2018, we were unable to access the Groups API and parse Facebook seller posts. With modifications, we were able to integrate Amazon and Labyrinth information. Later in this document, we discuss challenges related to data collection and integration.

B. Deadlines

We defined the following milestones in the design document submitted right before spring break:

- **March 17th** - Have a scaffolding app deployed on Heroku.
- **March 24th** - Consolidate information sources (Blackboard, Facebook, Amazon, Labyrinth).
- **March 31st** - Implement basic functionality, including book search.
- **April 7th** - Implement Facebook Textbook Exchange functionality (reach goal).
- **April 14th** - Alpha Test. Fix bugs and implement suggestions.
- **April 21st** - Beta Test. Fix bugs and implement suggestions.
- **April 28th** - Public release (course deadline for alpha test was April 27).

We immediately faced obstacles meeting our second and third milestones due to difficulties in obtaining a consolidated coursebook list, and delayed responses to our requests to obtain API access to Amazon and Facebook. All three sources eventually turned us down, and we set out to collect data using web scraping.

By what was originally our alpha test deadline (two weeks before the actual alpha test), we were able to scrape most book data, process and POST it to the database, and implement basic search functionality. From here, we decided that an in-app textbook exchange system was a necessary core feature, since we could not use the Facebook Groups API, and the current user experience on the

Facebook Textbook Exchange group leaves much-to-be-desired. Namely, the user is unable to see what books are already sold (unless if the seller was diligent enough to delete the post), and consolidate all posts related to a class without scrolling through months of posts. The result is a lot of messaging and waiting time - things that can be eliminated with a well-designed exchange system.

The in-app textbook exchange system was relatively straightforward to implement, because our database schema was designed with the possibility of adding this in mind. Fortunately, because we had not planned any feature additions during the alpha and beta test weeks, we were able to work on the exchange system and create a fully-featured app by the beta test.

This built-in flex time ended up being essential for ensuring that we had a polished final product. Even if the flex time hadn't been needed, it could have been used to build additional stretch goal features. For this reason, **we recommend that future COS 333 groups plan their schedules leniently** to allow for delays created by factors outside of one's control.

II. Implementation

A. Front End

React

React is a front end framework that abstracts portions of the webpage into components. We chose React because its *one-component-one-file* paradigm makes it easy to scale applications and understand how individual UI components work. In traditional front-end programming, content, style, and interactivity are handled in separate files, and each file may contain information for multiple pieces of the website (i.e. a single CSS file might contain styling for an entire webpage). This makes it difficult to find exactly what file to edit when making a change to a specific UI component. We loved React because of its modularity despite the steep initial learning curve.

React divides an app's components (e.g. the sidebar, a button, etc.) into individual files, and keeps the styling and logic together, which makes it exceptional at prototyping. The developer knows exactly where to modify a component and doesn't have to worry about his/her changes affecting other components of the webpage.

Client-side Computation & React Router

Some components, such as autocomplete, would be sluggish if they had to wait for a server response(s), so we performed most computations on the client-side. Additionally, our app uses *React-Router*, which enables client-side routing. This means that even though the URL path changes depending on the page viewed, the server only has to serve a single page. This helps keep page switches in the app fast while sacrificing a little bit of initial load time.

Redux

Redux provides us with a global datastore and state-controls the app, enabling separation of concerns between retrieving and maintaining consistent data, and component rendering. This minimizes messy connections and potential data inconsistency across components. At the core of Redux is a suite of services that interfaces with the server and updates the state accordingly, but is opaque to the component rendering process. While this seemed like a good idea in theory, in practice, it made the codebase hard to understand for those new to Redux and was a barrier to adding features, because Redux required many file changes in order to implement new features.

Material UI

We chose to use *Material-UI* for layouts and simple components, which gave us a modern, minimalist design. It offered many premade base components that we could build on top of (including textfields, data cards, sidebars, etc), and made styling a breeze despite our minimal combined experience in UI design.

B. Data Pipeline

Scraping Blackboard

Scrapy is a free and open source web crawling framework, written in Python. We used Scrapy's API to aggregate data from Blackboard, Labyrinth, Campusbooks, and Bigwords. Upon scraping required readings from Blackboard, we wrote some scripts to process this Blackboard output and create data structures that model the *classes and books* tables in our database. Many fall courses did not yet have required reading, so to be consistent, we decided to only include courses from the spring. It was fairly trivial to format the data in a way that was consistent with our *classes* and *books* schemas, since no price or description information is contained within these relationships.

Scraping Retailers

Populating the *listings* table was quite difficult, because we needed to scrape third-party sites (Campusbooks and Bigwords) to get descriptions and images, instead of pulling these using Amazon's API. We used the ISBNs of textbooks to find the correct pages for scraping third-party sites. However, this was limiting in some ways because many sites have tools that are designed to prevent scraping (URL tags and ID numbers). As such, we were limited in the sites that we could scrape.

Significant additional processing needed to be performed on Campusbooks data: the purchase links on their webpages are affiliate links that redirect after a timed delay to Amazon. For example, this intermediate url to this Amazon url. Because delays may deter users of the application, and displaying Campusbook's links would give the false impression that our app is monetized, we needed to recover the true Amazon link. Since Campusbooks used a meta refresh instead of a traditional 503 redirect, we could not directly recover the destination URL using a HTTP library

(such as Python's *requests* library). Instead, we had to download the webpage using BeautifulSoup, recover the Amazon link from meta tags, and remove the affiliate tag.

Finally, once all JSON objects were created, we populated the production database by sending a POST request.

C. Back End

The backend consisted of a server running *Node* with a *MongoDB* database. Node is a Javascript backend framework, which we chose because it was in javascript, enabling us to reuse the same object representations on both client and server-side and kept the codebase in one language.

Database

Our database was powered by MongoDB, chosen primarily because it was easy to map MongoDB documents to javascript JSON objects. After careful thought we decided to represent all data with five collections (classes, books, users, listings, offers). To minimize the chance for data corruption, we use only one way bindings, and have child entities hold references to their parents, rather than the other way around. This proved extremely helpful as it was easy to create and delete entities since their parents did not need to be modified. For instance, an offer could be created for a book and a listing, and after it was accepted, it could be deleted without leaving a trace.

We did make a mistake of having the users hold a list of books they were selling, which is information that could have been queried through the listings. This proved to be an issue because when a listing was deleted, it meant that the selling user also had to be updated, and on at least one occasion this caused consistency issues.

API

The API provided endpoints to access and modify these collections. We used *Express* and its powerful middleware system to implement the endpoints. Any action that involved a user passed through a user authenticated middleware that accepted a token, ensuring that only that user could perform the action. All API requests and responses were then serialized using json-api-serializer, which adhered to the JSON API format, ensuring that all data passed between the client and server was in a uniform format.

Testing

The API endpoints were lightly tested using *Mocha* with *Chai*. We used the testing primarily to ensure that the API endpoints remained functional after any changes. By guaranteeing the functionality of the API endpoints, we could ensure that any discrepancies that surfaced on the client were most likely due to client-side code and not server-side code.

D. Deployment

Our app was deployed on *Heroku*, chosen for its simplicity of use. We used the *mLab* add-on to provide the MongoDB database. Since Facebook now requires all apps using the SDK to run on HTTPS because of the privacy debacle, the app automatically runs over HTTPS, which is provided out of the box by Heroku. Surprisingly, the entire deployment of the app was seamless, and we did not face any issues in this area.

III. Difficulties and Challenges

A. Obtaining Booklists

From our experiences at Labyrinth, we knew that a centralized listing of books existed, since Labyrinth representatives at the store always pull up a document containing every course and its books. We contacted Labyrinth multiple times asking for this listing, but never received a response. We then contacted the Registrar and the Office of the Dean of the College, and were told that they were unable to assist us due to their contractual relationship with Labyrinth.

Fortunately, our contingency was to scrape the information from Blackboard, which was possible because a "Reading List" section exists for each course. Since each course page is publicly visible, even to students not enrolled in the course, we were able to scrape all course information from Blackboard using *Scrapy*, as detailed above.

B. Amazon Integration

Next, we needed to get price, images, and description information, and this is where we faced the most challenges. Our first task was to obtain book information from Amazon. We wanted to tie each book to an Amazon record to ensure data consistency, and also so that we could arbitrarily pull as much information as we needed about book information or prices on Amazon. Many websites (like camelcamelcamel.com) use the Amazon Product Advertising API to provide price comparisons, so we had every reason to believe that our project would similarly be able to get prices in real-time from Amazon.

However, Amazon was adamant about not granting us API access. After three failed reviews of our production-quality app, we decided to pursue other means of obtaining this information as described in section 2B.

C. Facebook Integration

Our next major feature was to integrate with the Facebook Textbook Exchange, the current forum within which students exchange textbooks. In February, we obtained admin access to the Facebook Textbook Exchange group, and were able to use the Groups API to programmatically get posts upon

supplying an access token. We could then process the posts and extract information about the books being sold.

However, this all changed on April 4th, 2018 when Facebook temporarily closed off access to its APIs in response to the Cambridge-Analytica scandal. At the time of this writing, Facebook is still not accepting applications for app review. Thus, we were unable to get posts from the Facebook Textbook Exchange group, since Facebook blocks scraping attempts.

D. Obtaining Book Listings *sans* Amazon

Scraping Amazon also did not work, since we found that our script would consistently fail after reaching a certain point. We suspect that Amazon was either doing A/B testing or rate limiting upon detecting behavior consistent with scraping. The reason why scraping stopped working was that the structure of each page physically changed inconsistently in ways which were impossible to predict. As a result, we scraped third-party sites already using Amazon's API.

Almost all our issues were due to unexpected issues with external dependencies. Despite these difficulties, we were able to find largely workable solutions. This experience was highly informative, and in the future, we will know to set higher thresholds of confidence for whether or not an external dependency is likely to succeed.

IV. Future Work

As mentioned in the previous section, we were not able to obtain API access from Amazon or Facebook. Thus, in order to provide truly dynamic prices in real-time, we would have to run a process on the server and have it feed new data directly to the database.

The pipeline takes 12 hours to update information from Blackboard, Labyrinth, and Amazon. Fortunately the information from Blackboard and Labyrinth is fairly stable, however Amazon updates its prices often. Ideally we would like to be able to capture Amazon's frequent price changes. One idea is to develop a method that detects when a page is updated, so that we only scrape the books that need to be updated instead of the entire collection.

In terms of additional features, camelcamelcamel.com is a site which provides historical prices for Amazon products. Adding price histories to our product could prove to be a huge value-added as students could then be advised on the optimal time to buy/sell their books to maximize financial gain.

V. Lessons Learned

The most important lesson we learned was that core functionality should never rely heavily on external dependencies. Our original app hinged on obtaining data from Blackboard, Amazon, and Facebook groups; however, as mentioned in the difficulties section, every single one of our dependencies fell through and we had to adapt our app.

We also learned never to commit API keys. We were suspended by Sendgrid because we accidentally committed the API key. Even though we quickly caught the error and removed it, it remained within the commit history and after the repository went public, it was discovered by the Sendgrid API key detection bot. Through this process we learned how to use the git reflog and filter-branch commands to tinker with commit history, and to never, ever commit API keys.

Use stable, popular libraries and don't rely on libraries maintained by only one or a few people, unless they are for something extremely simple. While having our API conform to the JSON-API specification was a good idea in theory, in practice it was a nightmare because the serializer library we used, json-api-serializer, had many corner case bugs. Some of these bugs were fixed in the repository but sat in untouched pull requests because the original maintainer of the repository was not responsive.