David Fan (Leader) Roland Fong Nathanael Ji Kyle Xiao

TigerTexts

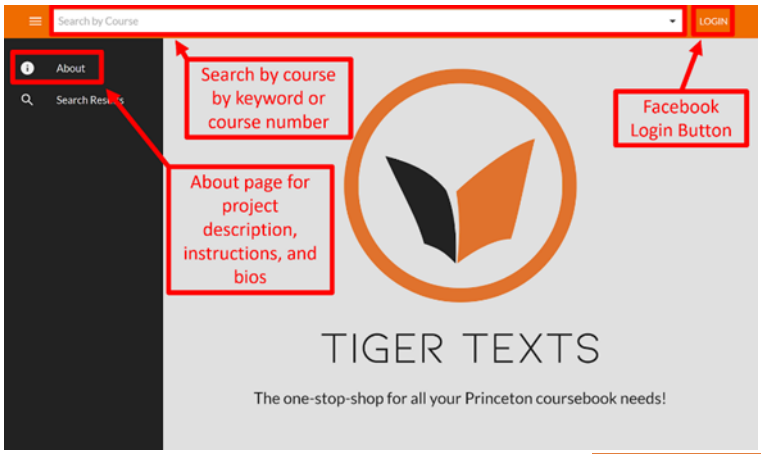
Product Guide

COS 333 Spring 2018

User Guide

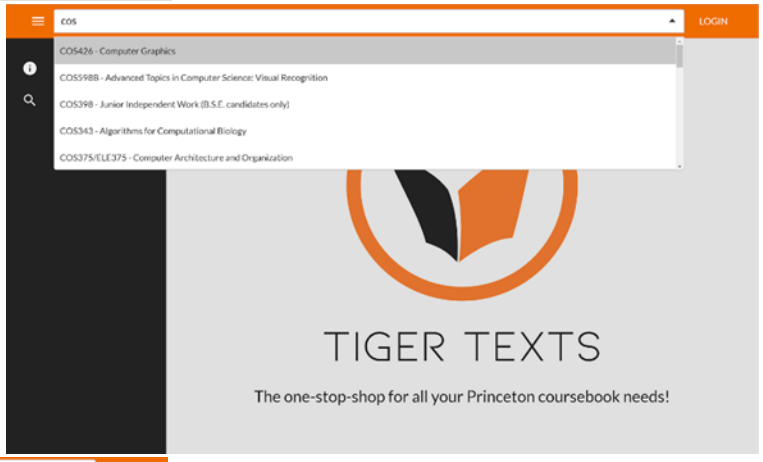
TigerTexts addresses the problem of how students can efficiently get coursebooks at the best possible prices. When buying coursebooks, students have to simultaneously consider online providers like Amazon, Labyrinth, and student hand-me-downs, in order to make a well-informed decision. Gaining perfect 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 providing a sleek all-in-one interface that displays the required textbooks for each course. For each book, **TigerTexts** provides prices and purchase links from various sources including our in-app exchange platform designed to supercede the Facebook Textbook Exchange.

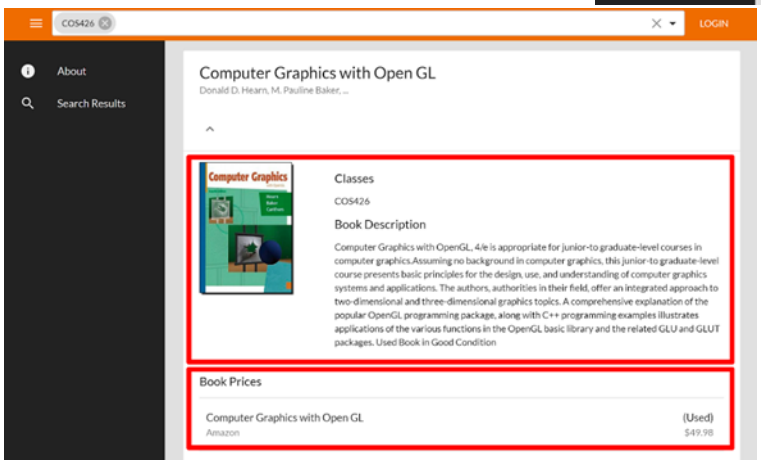
I. Public Features



Starting from the landing page at tigertexts.io, the user is greeted as a guest. From this page, the user can search by course in the **search bar**, examine the **About Page**, or **login through Facebook** for user specific features.

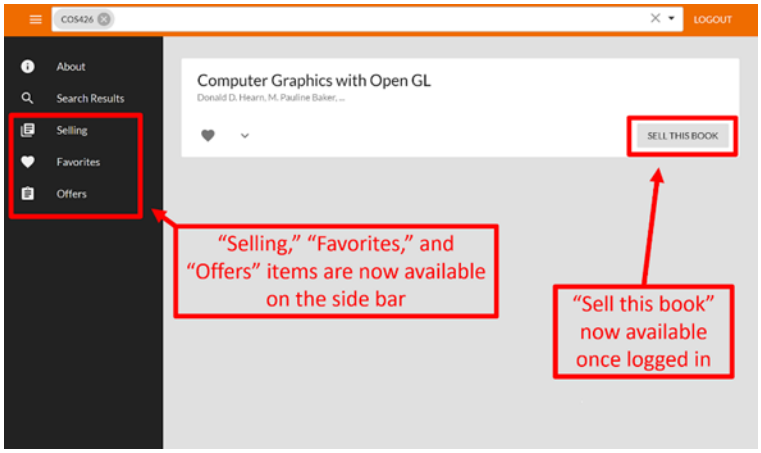
As the user enters terms into the search bar, an auto-suggestion window will appear to facilitate selection of the correct course. When searching, use of both course number (e.g. "cos426) and course name (e.g. "computer graphics") are acceptable.





For each book in the search results, the user can click the drop-down arrow to reveal more information. The user can then view book descriptions as well as the sellers and offering prices. However, guest users cannot make offers or sell books until they log in.

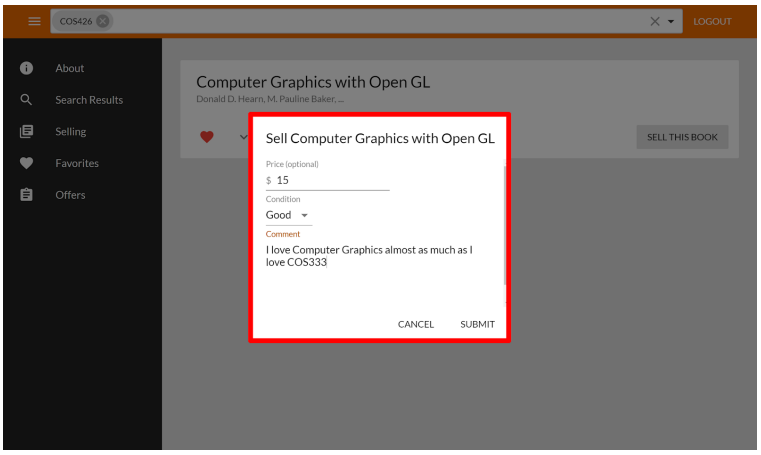
II. Logged in User Features

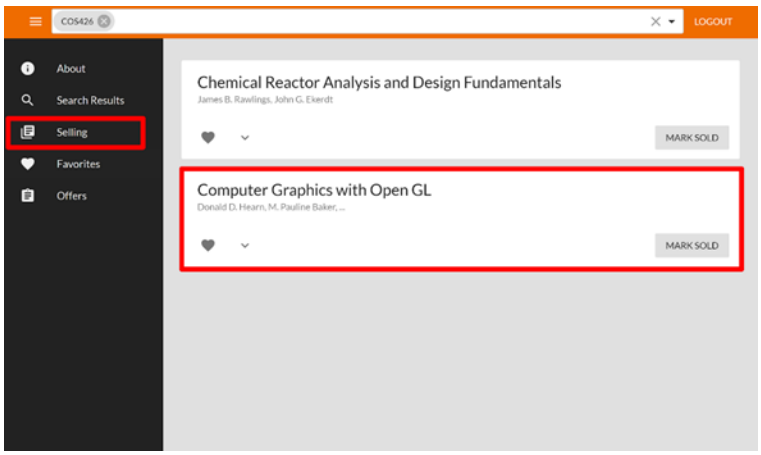


Clicking the "Login" button at the top-right corner of the screen allows the user to login through Facebook. This gives the user access to more functionality within the application, including the "Selling," "Favorites," and "Offers" tabs on the side bar.

A. Selling a Textbook

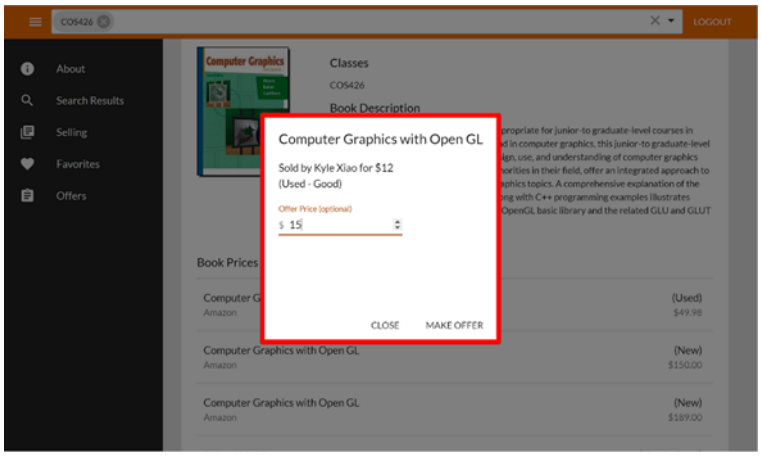
Choosing to sell their book, the user can enter a preferred price, the condition of the book, and any touching comments they'd like to add to their post.





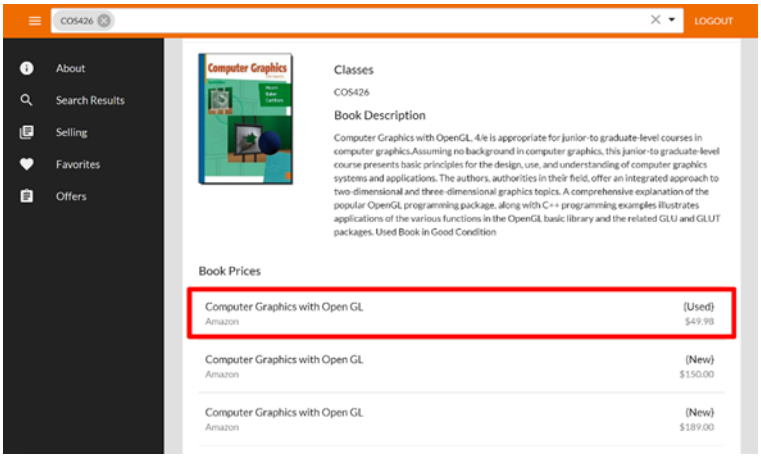
Now that the user is selling this book, the user can go to the "Selling" tab on the side bar and view all of the books they're currently selling. Once a buyer has been found and the transaction is completed, the seller would mark the book as sold. If the seller does not do so within a few days, an email reminder will be sent.

B. Buying a Textbook

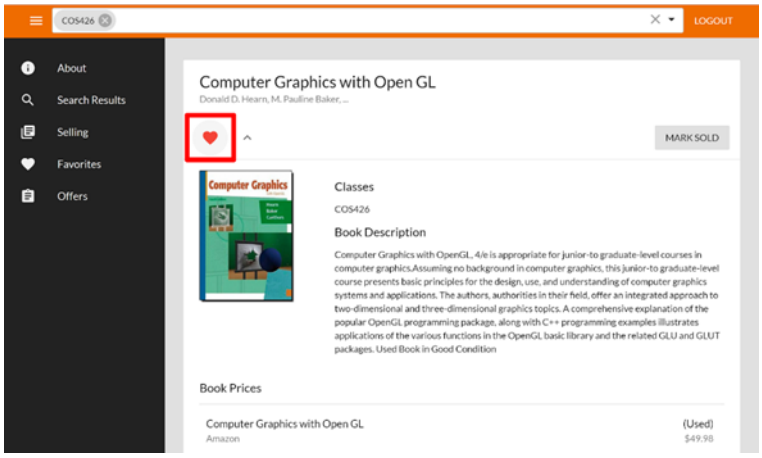


On the other hand, if the user wants to buy a textbook from the sale post of another student, the user can enter an Offer Price, or their bid for the item. If the seller accepts, an email notification will be sent to the seller via *Sendgrid*, so that the transaction can continue.

Once the user has made an offer, they can see it (and all their other offers) presented in the "Offers" section of the sidebar. Should a user decide to revoke their offer for a book, they can very easily do so by clicking on the "Revoke" button. Sellers have the option to accept or decline an offer, which then sends an email to the buyer.
Alternatively, the user can choose to buy through a third-party retailer like Amazon. In that case, the user will be redirected to the retailer's page for that textbook.



C. Favoriting Books



If the user wants to keep track of a book for future purchases, they can add it to their "Favorites" list by selecting the heart icon below the book's title. To view their Favorites list, the user can simply select the "Favorites" tab from the sidebar.

Developer Guide

Data Collection and Processing

We first use **Scrapy** to scrape Blackboard and identify the required textbooks for each course. Because we did not have access to Amazon's Product Advertising API, we needed to obtain pricing and book data by scraping. As it turns out, scraping Amazon doesn't work because the structure of the page changes on a regular basis, and they are able to rate-limit users. Our solution was to scrape third-party sites that already have access to Amazon's Product Advertising API, and have simpler page structures. Specifically, we scraped campusbooks.com to obtain prices and purchase URLs, and bigwords.com to get book descriptions and high quality cover images. As a local textbook source, Labyrinth was also scraped.

To run the scraping pipeline, first run Scrapy inside `/scripts/blackboard_crawler`, following the README file. Copy this Blackboard JSON output to the `/scripts/labyrinth_crawler`, `/scripts/amazon_scraper`, and `/scripts/description` folders and run Scrapy again in each. If possible, run these processes concurrently (or on multiple machines and transfer files after-the-fact) to reduce overall runtime.

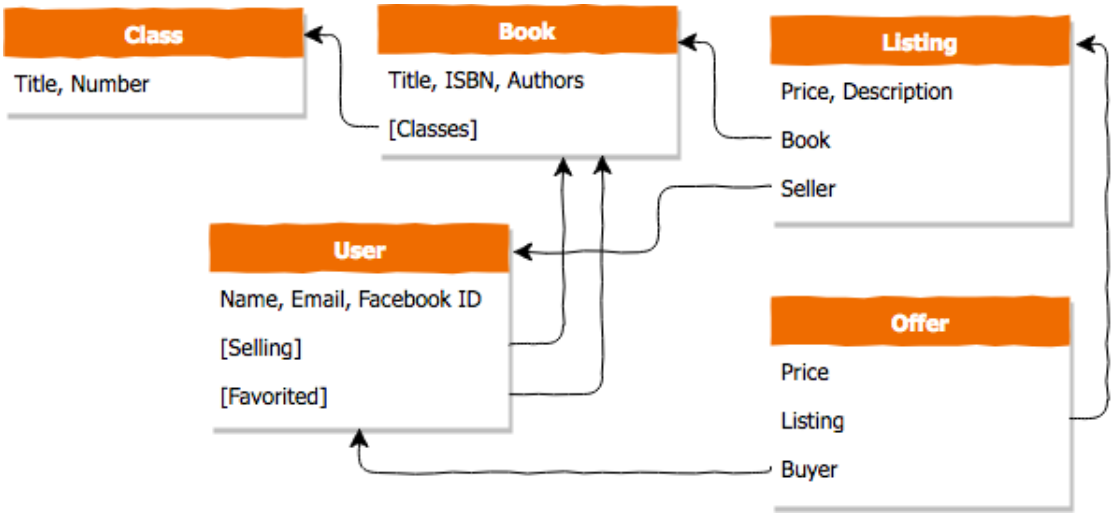
Then, drop the *classes, books,* and *listings* tables in the database if they already exist. To populate the local database, run the Python scripts *post_classes.py, post_books.py, and post_listings.py,* **in that order**, with the argument "local". For example, to populate the classes table locally, one should run `` `python post_classes.py local` ``. To populate the production database, do the same but with the argument "prod". If the local *listings* table is already up to date, one can run *local_to_prod.py*. This script simply reads the local *listings* table and POSTs the JSON object directly to the production *listings* table, which is significantly faster than running `` `python post_listings.py prod` `` since extracting price urls (by downloading each CampusBooks link with BeautifulSoup and examining the meta refresh tag) is the main computational bottleneck. However, if the local database is not up-to-date, and one does not wish to update the local database, one should just run `` `python post_listings.py prod` `` directly and not use *local_to_prod.py*.

Backend

The **Node** backend server is responsible for handling query logic and serving resources via an API interface. All code is written in **Javascript ES6**, which brings a wide variety of convenient syntax and language features that help make the code more readable and maintainable. The most interesting of the features that is widely used is the async-await paradigm, which enables us to avoid so-called "callback hell" and really long "`.then()`" chains associated with Javascript promises, which are deferred computations.

Database

The **MongoDB** database contains five collections, summarized in the diagram below. Two particularly special collections are the Listings and Offers. Listings represent a listing on Amazon, Labyrinth, or on the app for selling a book at a certain price. Offers represent an offer that an interested buyer makes to a seller's listing (assuming it is an in-app seller).



Notice that the database objects have one way relations that are *additive*, that is, objects lower down on the list contain or depend objects above them, rather than vice versa. This is to ensure that we can easily add, for instance, a listing, without having to modify books, which reduces the chance for error. Additionally, it makes it easy to delete items lower down on the list as they are more ephemeral, for instance, offers are deleted as soon as they are accepted, but this operation doesn't affect books or users.

API

The client accesses this information via a RESTful API. Any user specific portions of the API are authenticated by requiring a token from the client. The primary endpoints are as follows:

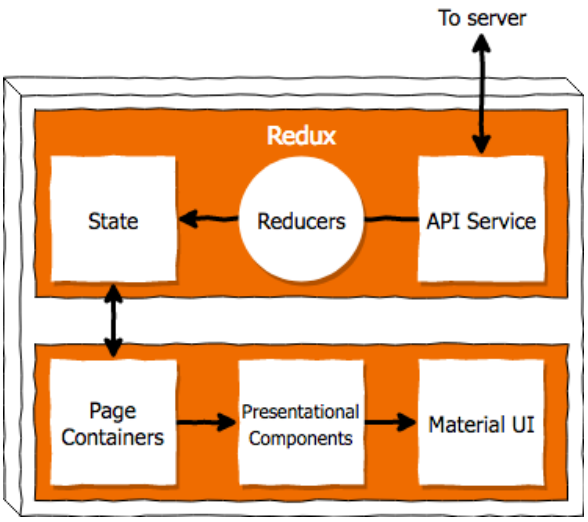
Endpoint	Auth?	Description
`/api/classes`	No	CRUD operations for classes. Currently the client only uses GET to fetch classes; the remaining operations are used to input scraped data
`/api/books`	No	CRUD operations for books; the client also does not mutate the collection
`/api/listings`	No	CRUD operations for listings. The client uses all endpoint operations
`/api/users`	Yes	Allows login and any user specific operations. This enables the client to edit user favorites and selling or making offers for books. All modifications to offers objects are done via this endpoint since offers are always associated with users

The API returns responses in JSON API format. This is important because this gives us a format to return not just the data, but also associated metadata or included resources. For instance, we may choose to return a book but only optionally return its associated classes in the event we wish to save on the side of the API response. Though not currently utilized, the specification also provides room for pagination among other things. We use `json-api-serializer`, a third party library, to serialize and deserialize API responses and requests. This is also used on the client.

Auth Flow

We use Facebook login for our app instead of CAS authentication, because we initially wanted to integrate with Facebook and parse posts made in the Textbook Exchange group to generate listings (and auto-post to the Facebook group when new listings are made on the website). This may become possible in the future after the dust settles down on the Facebook-Cambridge analytica scandal, which is why we left this feature in. After a user logs in, we generate a JSON web token (JWT) and store it as a cookie in their browser to remember their login session. Every time a user endpoint is called the JWT is sent in to authenticate the request, ensuring that random users cannot impersonate other users (at least without an active token).

Frontend



The app is developed in **React**, a component based front end development framework, along with Redux, which provides a data store, and React Router, which allows for client side routing. All code was written using the latest ES6 syntax, which was provided out of the box by React.

The client code is organized in the following manner:

- Redux Actions and Services - These are responsible for interacting with the server, and requests are made with the built in Javascript `fetch()` method.
- Redux Reducers and Store - The core client side logic of the app is handled within state reducers that determine the state of the app, including user information, book lists, and offer lists.
- React Router pages - Despite having a single container page that houses the entire app, complete with a nav bar and side bar, there are several pages that display data. This includes the Book list container (route `/books`), the about page (route `/about`), and the offers list (route `/offers`). Each link on the sidebar routes to one of these pages, which are redux aware and so read from the redux state.
- React components - Each container page contains multiple components which are responsible solely for the presentational rendering of the app.

Finally, components were rendered with Material UI which allowed us to do relatively little styling.

Development, Testing, and Deployment

The app uses the **yarn** package manager and comes with scripts for easy development. The scripts allow you to run a development version of the server and the client on its own server, while a production build can be created and served via the server locally. All code is linted for consistent style with **ESLint**, and the server API endpoints are lightly tested with **Mocha** and **Chai**. After committing to the main repository on **Github**, the app is deployed manually via **Heroku**. DNS is managed with **Namecheap**, allowing our app to be accessible at tigertexts.io.