PTX2 Report

Overall, we are very happy with how the project turned out. The interface is sleek, simple, and easy to understand, and, through the power of Bootstrap, is quite aesthetically pleasing. The core functionality and improvements over the existing PTX that we envisioned for the project—the dynamic searching, Blackboard reading list integration, and new inventory management system that we outlined in our elevator speech—are all satisfactorily implemented, and although there are a few additional features that we did not manage to include in the website, these were mainly luxury features that would not have added as much to the user experience, or even detracted from its simplicity.

Milestones

In terms of milestones, we realized fairly early on that the ones we had laid out in our design document were going to be quite different from what we would actually do, especially in terms of the order in which we did them. That being said, a lot of the preliminary work and set up went as planned. For example, our first major milestones were to set everything up in terms of creating a Github project, Django site, and Heroku app. These culminated in getting our app to behave and display “Hello, world!”. As brand new Django and Heroku users, we were quite proud of being able to download all the packages and set up everything in Heroku, even if it wasn’t a very impressive feat in of itself. At the same time that we did this, we also created the mockups of our site and decided on what we wanted it to look like, which made the coding of the front end much easier.

Our second milestone was completing the scraping. This was done early on, which was a smart decision, because it took even longer than expected and had more issues than we anticipated. The first major hurdle we overcame was being able to login to CAS via terminal, which was needed to access the crucial Reading List material from the Blackboard pages. We finally fixed this issue by storing cookies and establishing a session through use of the Python CookieJar package. The next step was to actually scrape from Blackboard and Amazon, who provided adversity in their inconsistent source pages. This was the toughest and most time consuming part. Finally, after a lot of testing and rewriting functions, we were able to scrape all the data we needed into one comprehensive file, which we named finalcopy.txt, aptly, and convert that to JSON. This was eventually uncoupled from the population of the database, because we realized that we would likely need to restructure the database, and we didn't want to populate the database and then need to erase everything shortly after.

The third milestone was the coding of the front end, creating the website in HTML and CSS. Using the mockups we had created earlier, and the very easy to follow Bootstrap documentation, the initial website was rather easy to code. However, this milestone was extended to last essentially the entire process, as the structure and look of the website continued to evolve as the features of the site became more and more concrete.

Another important milestone was the aforementioned restructuring of the database. This included the updating our models to include PhysBooks and Listings, which allowed for the existence of physical books that were not being sold, and the current “Books Owned” section. Users said that they enjoyed being able to own books, to sell them extremely easily later, and simply to use as an organizer to see what books they have (in conjunction with courses added in the sidebar - so they could see how many books out of the books required they had).

Adding and removing courses was another milestone, because those were a few of the first major functionalities we added. Also, the adding and removing courses search feature was borrowed heavily in searching for books and courses in the top search bar. This gave us insight into how to connect a view function with a form located in a modal window. A lot of our other features also included modal windows, so adding and removing courses was the example we worked off of for many other features.

The last major milestone was selling and buying. Given the knowledge and experience we had gained from the previous few milestones, this ended up being quite easy. Creating a listing was similar to the creation of a new book when we populated the website, and the moving of a book between different sections of different users' profiles was similar to adding and removing a course.

There were a couple of milestones in our design document that we ended up deciding to leave out. These were mainly features that we had mentioned in our design document as nice to include, but were a bit more complicated to get right than we had anticipated. The first milestone we decided to omit was the "Sell All" feature, through which at the end of each semester, you would have the option to sell all of your currently owned books. This would have been a nice feature to further cement PTX2 as a book organization/inventory tool, setting it apart from the current buying-and-selling centric PTX, but ended up being difficult to get right. For example, a rudimentary implementation of this feature would have been quite simple, essentially just generating a bunch of sell forms as per how we currently do when a user clicks on the "Sell" button of a book they own. However, this would not be much of an improvement over having the user manually indicate they wanted to sell each of the books in their bookshelf, and so would need some further consideration before it would be worth implementing.

The other milestone that we decided to be impractical was the "Fast Buy" option, wherein a user would be able to quickly and easily buy all of their needed books from the lowest student seller. This would make the buying process much closer to going to Labyrinth, presenting them with your course list, and walking away with all of your needed books, which was something we strove to emulate with PTX2. One problem that arose with this feature was that this would have likely caused a lot of collisions with multiple people trying to purchase the same listings, so we would first have needed to implement a better way to deal with these sorts of collisions. Additionally, this type of one-click purchase could be dangerous, especially when we really have no way to enforce that transactions are completed, so in the end this feature was left out.

What was your experience with design, interfaces, testing, etc.?
The design of the user interface was one of the more successful parts of the project. From the very beginning, we made mockups of the main pages of the site, so we knew exactly what we wanted to achieve in terms of look. In making the mockups, we also consulted many existing similar applications and sites to get ideas as to how to make the site most user friendly and intuitive to use. For example, we consulted Delicious Library 3, which possesses similar book management functionality, and this gave us the idea for the sidebar and bookshelf distinction. The navigation bar was somewhat inspired by easyPCE, as were the search bar and results page, and the book pages were variations of the existing PTX book pages.

With this elementary user interface in mind, it became a process of trying to replicate the mock up with actual HTML and CSS code, as well as slight tweaks to the navigation as we tested things out. In terms of the final look of the site, we decided that we liked the default Bootstrap

theme enough that we felt it unnecessary and almost detractive to impose the usual orange and black color scheme used by all of the TigerApp sites. We really liked how the minimalist design of the site in terms of both simple color and easy to understand text/symbols kept the attention on the main elements of the pages—the books.

One design feature that surfaced a bit later in the process was the extensive use of modal windows. We learned about this window type while using Bootstrap, and really liked how it reduced the amount of navigation that a user would do in a usual, everyday interaction with the site. Unlike PTX, where buying and selling takes you through a long series of new windows and makes it a long, drawn out process, the modal windows essentially broke down everything into a self-contained, separate thread, that allowed users to quickly return to the main tasks of browsing and managing their bookshelves. Especially since we wanted to make PTX2 more of a book/inventory management system, and less of a one-time buy and sell facilitator, it was important that the key elements of book management were easy and quick to complete. We also hoped that this would give the site a more polished and modern aesthetic, and make it more attractive to users.

Another aspect of the design that changed during the process was the eventual introduction of multiple ways to do the same actions. At the beginning, we were of the opinion that the cleanest and most effective design is one that is not redundant; essentially one where every action is only accessible in one way. For example, if you wanted to sell a book, there would be one clear way to do it: the “Sell” button on the book’s book page, as to not cause confusion. However, we also realized that this introduced latency into the completion of key features, and so we relaxed this condition, populating various pages of the site with many of the same buttons. For example, the ability to sell a particular book is now available 1) right on the bookshelf, 2) on the book’s book page, and 3) on the search results page. We found that making these types of key features more visible and readily accessible was more conducive to an intuitive user experience than our initial approach.

As for testing of our site, we did not conduct much formal user testing, instead using our own judgment and the opinions of friends that we showed the site to to determine what things worked and what didn’t. For example, the decision to make the navigation bar stay at the top of the page was a result of friend testing. Additionally, the aforementioned decision to include multiple copies of the same button on different pages of the site was motivated internally by the group’s members. Links to the profile in the top search bar, help and faq pages, and aesthetics of the site were more suggestions given by friends.

What surprises, pleasant or otherwise, did you encounter on the way?
We did not encounter many surprises throughout the process, though there were a few things related to Django that caught us a bit off guard. The main surprise that we encountered was that after turning off debugging, Django no longer serves static files, and leaves that up to the server. This was particularly surprising because we decided to turn off debug right before our demo (just in case we somehow ended up at an invalid page), and all of our CSS stopped working. Luckily, turning debug back on was a trivial process, and the demo still went smoothly, but it still incited a momentary panic.

Another surprise was that we learned that developing on Windows is very cumbersome and complicated. Alan, as our sole Windows user, in particular discovered that a lot of tools need to develop for Django and Heroku (gunicorn and foreman) were not supported on his machine,

and it was very hard to find particular substitutes to them in order to even launch the server. Second, in order to install a lot of the tools needed or the substitutes, we learned that you need either sudo or apt to install the tools needed, and of course, Windows also does not have apt or pip installed either… Finally, we all know that the windows command prompt is simply terribly written and does not have a lot of the basic commands that a command prompt should have, and is therefore also a detriment to developing in Windows. In response, Alan decided to install Linux Mint Petra onto his computer, which very suddenly made development a lot easier, given that the tools required to run Django servers is supported by unix environments. None of the other group members had this problem, so Alan was alone in his struggle, and ended up a bit behind during the first month of development by fiddling around with substitutes to python packages then finally giving up and installing Linux Mint in the middle of April.

This surprise was similar to the problem of the unexpectedly steep curve of actually downloading the software, installing it, and making sure that the settings were set correctly to allow the different parts meshed together to make one coherent website. One example where this forced us to change our design was our relation between the server, database, and Django. Our initial plan was to run our Django website on Google App Engine. Django's backend allowed us to run on a variety of databases, but in order to make Django work on Google App Engine, we had to configure our Django backend to interact with Google Cloud SQL as well as make it runnable through Google App Engine. It took a lot of fiddling around, but in the end, we could not even deploy the most basic Django website on Google App Engine, and then we started looking at other servers, namely Heroku. Heroku also took a lot of installation and manipulating settings, mostly because Heroku required PostgreSQL, but we finally managed to get our website up and running by pushing updates to Heroku via Git. The switch to Heroku ended up have a positive side effect, in that OIT had already whitelisted all Heroku applications, and made CAS integration a lot easier.

The final surprise that we encountered was more benevolent, which was how easy it was to have our website look nice just by using Bootstrap. We would highly suggest any team in the future to use Bootstrap, even if just as a base on which they would add their own customization.

What choices did you make that worked out well or badly?
In terms of design choices, the majority of our initial decisions were in the ballpark of what we used for the final website. Our use of Django was definitely a smart choice. The urls-models-views-forms framework was very compatible with our idea for a book exchange for several reasons. First, we needed many data structures to have relationships with each other in order to function. For example, a user can own a book, but not have it listed as selling, and thus a listing should have a relation with an owned book. Furthermore, we can have a book as an abstract class, i.e. its isbn, title, lowest price, but no user can own the book. Thus, each physical book should have a one-to-one relation with its abstract class, exactly like the idea of a class and an object in Java. This was very well supported by the ManyToMany and ForeignKey relationships built into Django. Second, we have many pages that have similar templates but we have to dynamically allocate the data when the user calls the specific page. For example, we have the course page template, the book page template, the pending page template, etc. By using Django template language to create variables that can be dynamically filled, and using views to find and fill the data to place in these templates, it becomes very easy to customize each template for individual (which is what our website is mostly about!). Along with this customization comes urls, where we can get variables from the url and pass it along to the views to specify which page to go to.

Later in the semester, we transitioned from the urls approach mentioned above to a different approach to minimize security flaws. We realized that if you simply entered in the url for a transaction id in which you were not the buyer or seller, you could still access that transaction page. To counter this, we implemented two security measures. First, we changed the url so it doesn't show the transaction id. Instead, we made a hidden value on the template page that is dynamically filled with the transaction id and then called by a GET request to obtain it. Second, we made it so that if you are not the buyer or seller of a particular transaction, it redirects you to another page, and says you cannot access the transaction.

Our decision to do extensive Blackboard scraping started out seeming like a very bad idea, especially with all the struggles we encountered in trying to get it to work. However, as it was an extremely important aspect of the core functionality of our site, we really tried very hard and devoted a lot of our resources into it. This ended up being a good decision, and we feel that it is a feature that sets us apart from both PTX and our "competitors", tex.

Another decision that went well was the use of Bootstrap, which really expedited the entire front end development process and just allowed for a really beautiful website.

One decision that did not go so smoothly was our initial database structure—or a few of the versions that came after that—and while everything worked in the end, it did cause a couple of hacky, roundabout fixes that could have been much easier given a better database. For example, we had to populate a user's selling list by looking through all of our listings, despite having a books selling field associated with each user. This was because we had somehow decided to make books selling a list of PhysBooks and not Listings, which would have solved the issue entirely. However, it was too late in the game too change, and wasn't too much of a problem anyway.

What would you like to do if there were more time?
Like the milestones that we actively decided to leave out, there were a couple of features that we would have liked to include, but were unable to due to time constraints. The first feature that we wanted to have was a watch list, through which you would get email updates when a certain price boundary had been reached for a book. For example, I could specify that I am only willing to buy a copy of a book if someone offers it for less that $40, and I would be able to get notified about the addition of such a listing through the watchlist. This was the first feature that would have still been a useful feature, as it would have been similar to the wish list implemented in PTX and the watching feature in the other PTX remake group, tex, but ended up being left out due to time. Even though this would not have been very difficult to implement, especially since we figured out how to send email notifications when you bought a book/one of your books was bought, it would have required another restructuring of the database, and given our time constraints and the number of other core functionality bugs we needed to fix, ended up being a larger undertaking than we were able to handle.

Another feature that we would have liked to include was an autocompleting search bar, similar to the one that is currently used for easyPCE. This was one of our initial major set-apart features, and it would have at least made the site look more noticeably dynamic. However, we think that our search results pages are structured well enough and are easy enough to navigate through that the autocompleting search bar is not as necessary as we initially thought it would

be. Also, most of the queries, as they will likely be for departments, will not be long enough for any kind of autocomplete to be that useful, so this wasn't the greatest loss.

A final feature that we had to leave out was reviews. We actually use reviews in our current version of PTX2, but only in a very superficial manner, using them as a defacto two-part transaction confirmation. There is definitely room for these to be used in a more constructive manner. For example, the current PTX has seller reviews, which give a very basic, numerical assessment of how good a seller is on keeping their promises with regards to a sale. However, as the system just uses a simple numerical value, it is still not very helpful. It would have been nice to do something similar to course reviews with our reviews, or perhaps including in various star fields, like with product reviews, where a buyer could rate a seller on promptness, accurate advertising, etc...

How would you do things differently next time?
We probably should have started coding together as a group much earlier in the process. Since we were all beginners to making any project outside of course assignments, we had a lot of learning to do about frameworks and downloading packages. At the very beginning, when we were all still trying to figure the various technologies we were using, it was a very independent coding process. As each member went through their own respective tutorials to learn how to use the different frameworks we were using, especially Django, a lot of redundant work was done. For example, as per the tutorials, each person made their own Django site, which, while important to learning how to use Django, established the development process as isolated to each person from the start. This made it difficult to effectively work on the various parts of the project together, which would likely have expedited progress.

It would have also been much more effective to have all used the same tutorials. This was particularly troublesome with Django, where between the four of us, we used three different tutorials, Tango with Django, Djangobook, and the official Django documentation. This meant that even though we all sort of knew how to use Django, the particular methodologies we have learnt were not completely in sync, and it occasionally made it difficult to figure out what our other team members were trying to do.

Another particular issue we had was with version control, and an overall lack of understanding about how to use Github. Thus, it took a while to get into the flow of committing and pushing changes, as well as remembering to pull. Many times we got unnecessarily about Git merging files, as the first time this happened, we somehow managed to completely delete our Github repository. An additional unforeseen factor that exacerbated this issue was that Michael initially used the Github desktop application, which dealt with the push and pull process in one "Sync," and didn't quite behave as expected. This combination of problems meant that a lot of changes went overwritten or accidentally deleted. For example, many times we perceived a bug fix we had done to not work, because it was not properly pushed into the repository. Other times it was more transparent what the issue was, but still as annoying. For example, one fix to views.py was overwritten at least five times. Things got more complicated when we started pushing to Heroku, as people would push to Heroku and not Github, which caused even more confusion about what version of the code was the most current.

Once we eventually got the hang of it, Github became one of our biggest assets. If one change made didn't work, we didn't have to go back to edit every file we changed (which was a lot even though it was only a couple lines in each file, just as a result of how Django works), we could

just reset head to the previous commit and work from there. This feature also helped us recover data we lost during a spurious commit. The takeaway from this whole ordeal is that we should have gained a more solid understanding of how Git worked and more firmly established a workflow for using it, which would have smoothed out the entire development process.

Another thing that we probably should have devoted more time to is figuring out how to properly develop with Heroku and Django. For example, only one member managed to figure out how to develop locally and the other three members decided to just push updates to Heroku. This meant each update took about 30 seconds, and just introduced unnecessary latency. Another problem with the database backend was that Django does not support migrations between models and live data. Whenever we made a change to models.py, which was we stored all the models (data structures), we would have to delete the entire database along with all the data, and then recreate the database with no data. This process was very tedious and painful, and thus we tried to make our database perfect every time we changed models.py, instead of incrementally debugging and making small changes. We did try using South, a database migration tool, but did not end up figuring out how to get it to work. This could be a useful tool for future groups to try using.

What should next year's class learn from your experience?
One of the most important things that we would tell next year's class is to really make sure the design document is well fleshed-out and the project is well-planned, especially for a project that involves a lot of user actions and database additions and removals like ours. We took a lot of time to write up the exact structure of our models, and although they changed quite a bit, they were still well-planned and acted as a good springboard. If we hadn't planned out our models well, that would have resulted in a lot of ripple effects, like the data we would have scraped, the setup of our bookshelf template, and more. Planning out our models relatively well saved us time from scraping over and over again for more information that we didn't anticipate using before (for example, if we had forgotten ISBN or something, which is super important), and from revamping our homepage over and over again. We also planned our models with the database actions, like selling, buying, adding courses, and marking things as owned versus selling them, in mind.

We also took a lot of time to read tutorials about Django before jumping into coding. We made sure to do things right the first time around, methodically editing urls, then views, then relevant templates. Without this level of care, we would have spent a lot of time just trying to figure out how Django worked at all, so taking a week or more just to thoroughly read about Django was a good time investment.

Next year's class could also learn from our experience that they should allocate a lot of time for possible errors, because things like the setup (especially PostgreSQL) and scraping Blackboard took much longer than we thought it would. Even though we didn't follow our initial timeline perfectly, our initial plan left a lot of time at the end, so we had enough time to finish our project.

Overall, we're very happy with the work we've done. We've all come a long way from just working on course assignments, to creating an entire project by ourselves. We hope next year's class realizes exactly how much they stand to gain from this project, and make the most of it! There's no other class quite like this one at Princeton, and there is no other project quite like a 333 project.