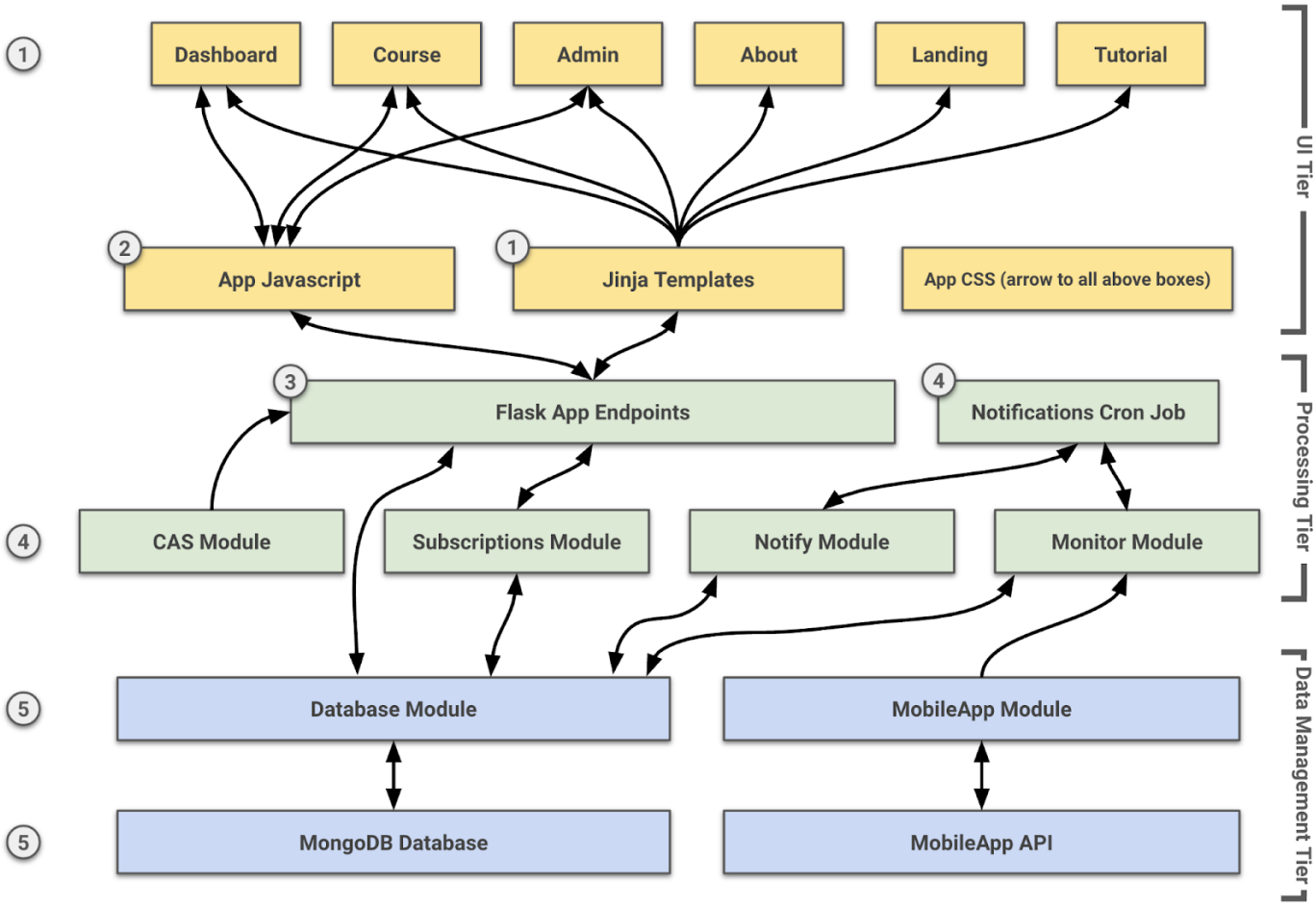
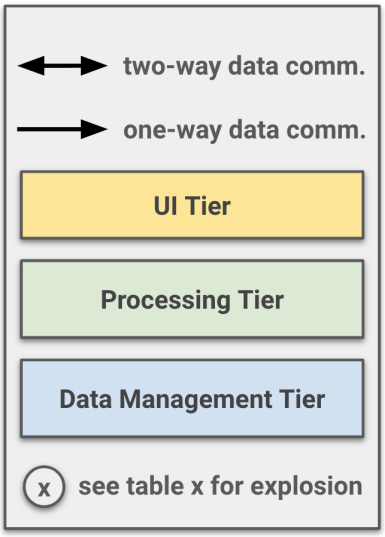
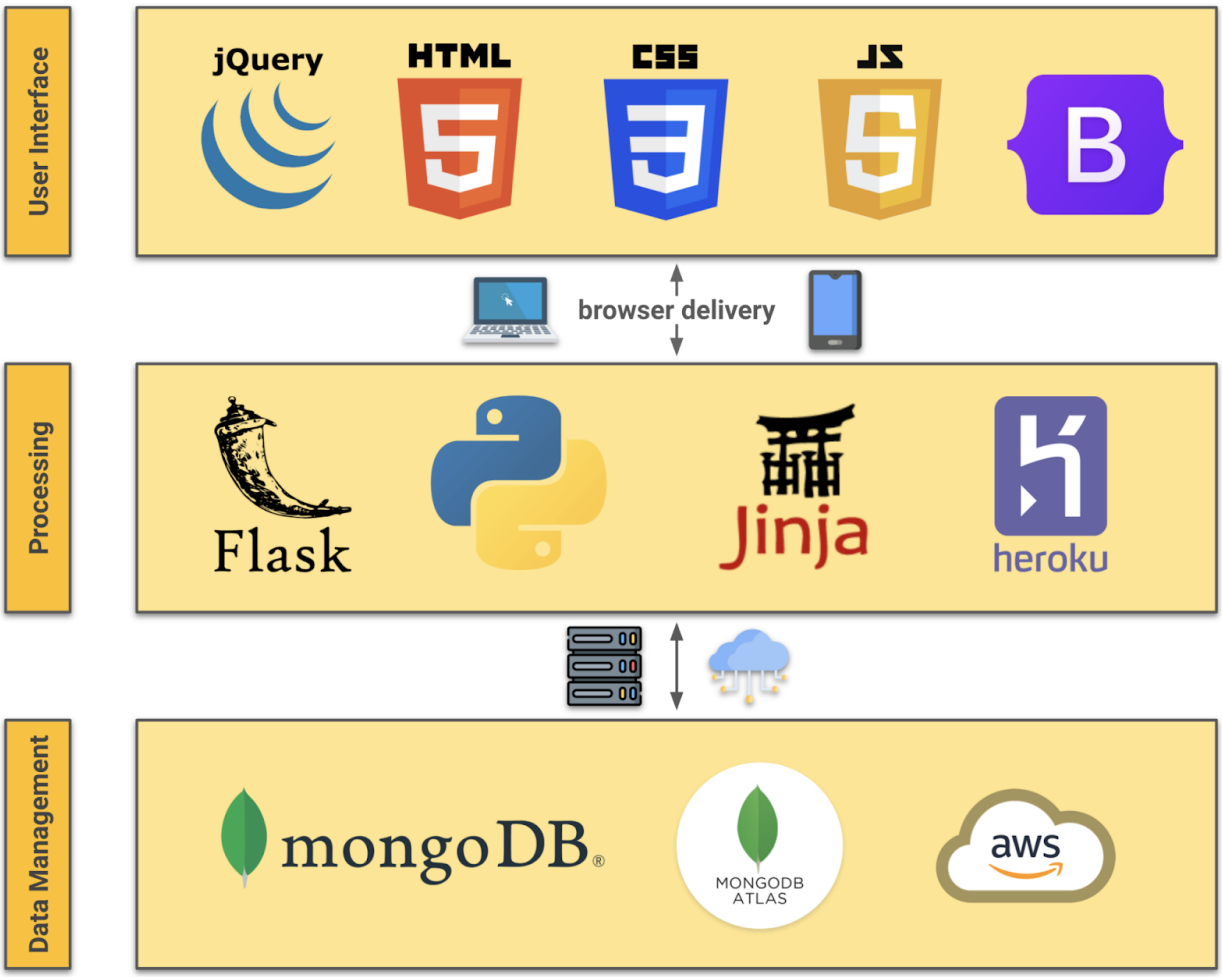
TigerSnatch: Programmer's Guide

Table of Contents

System Architecture



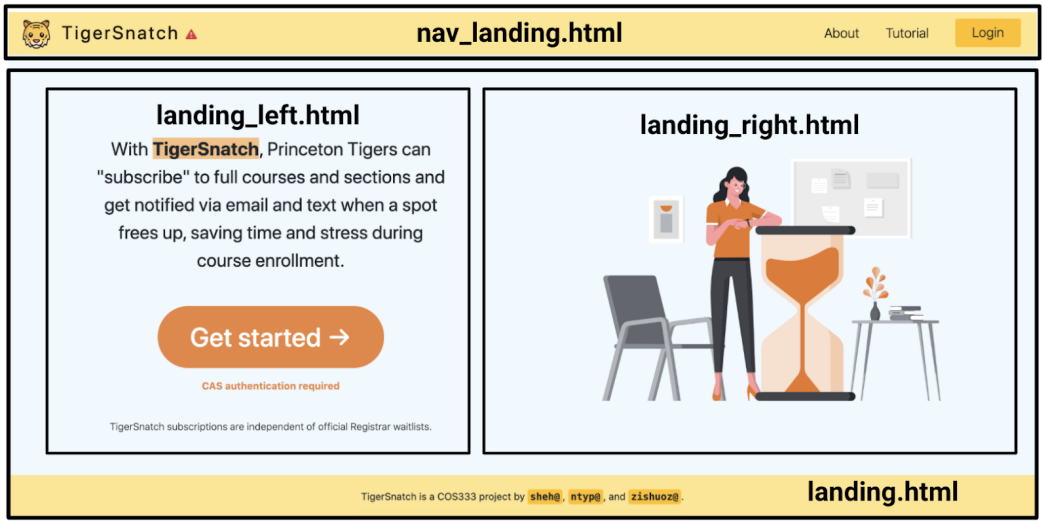
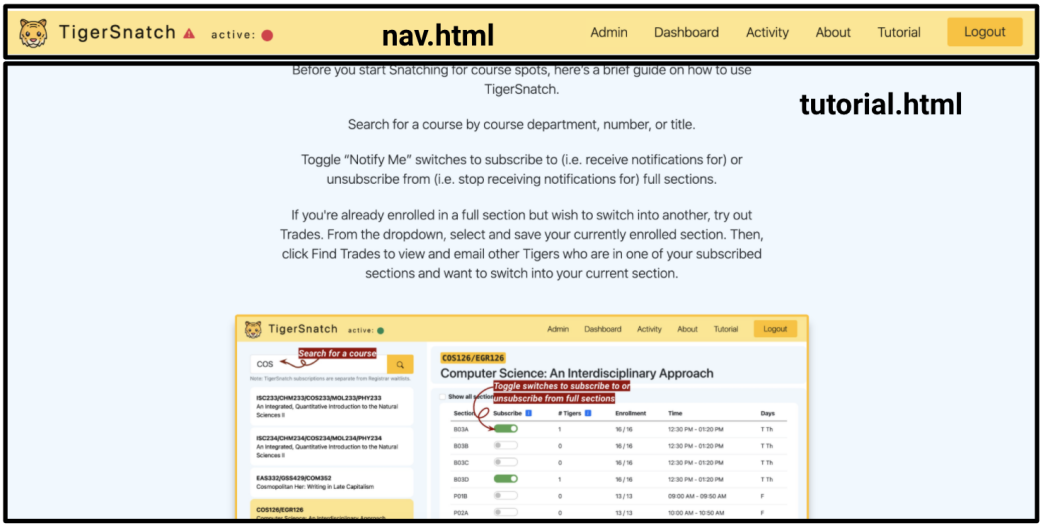


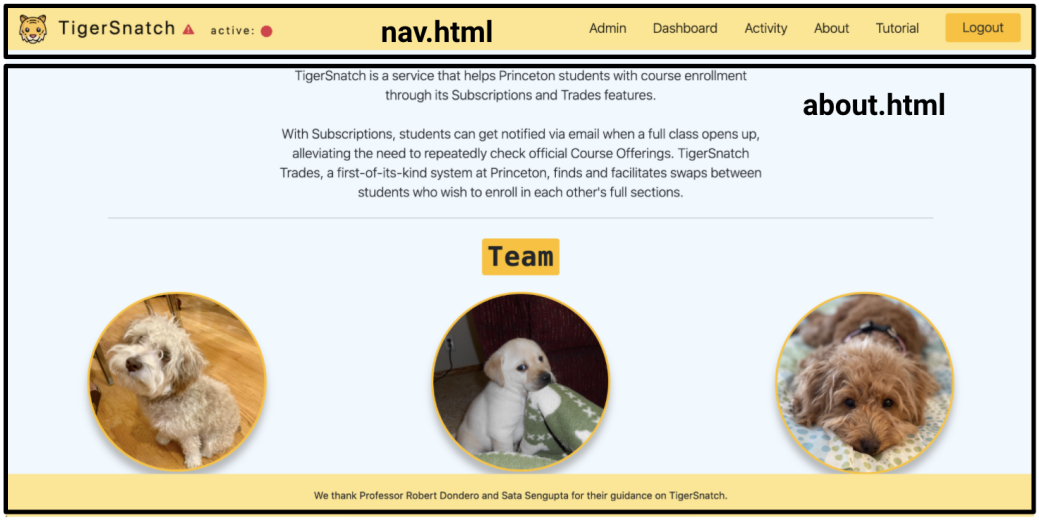


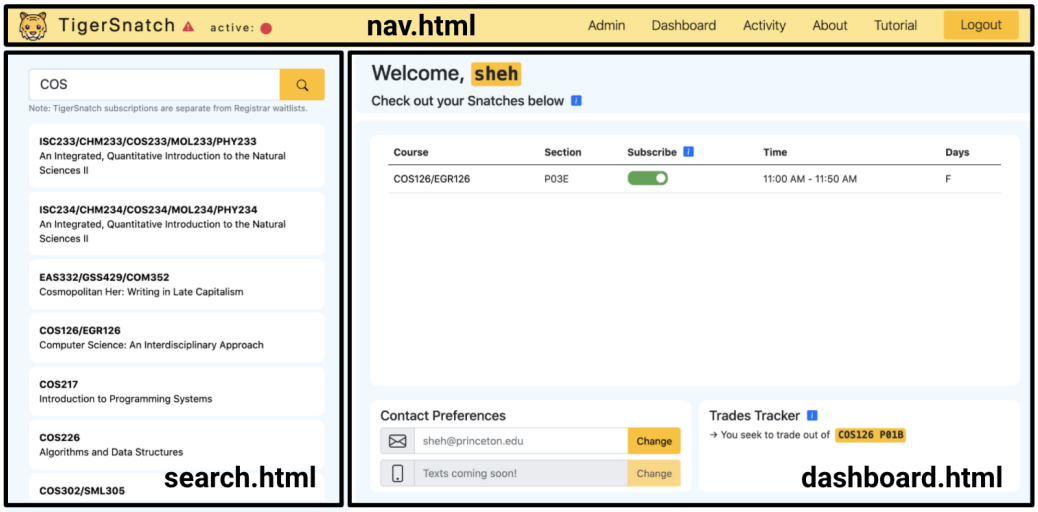
Technologies used in each system tier

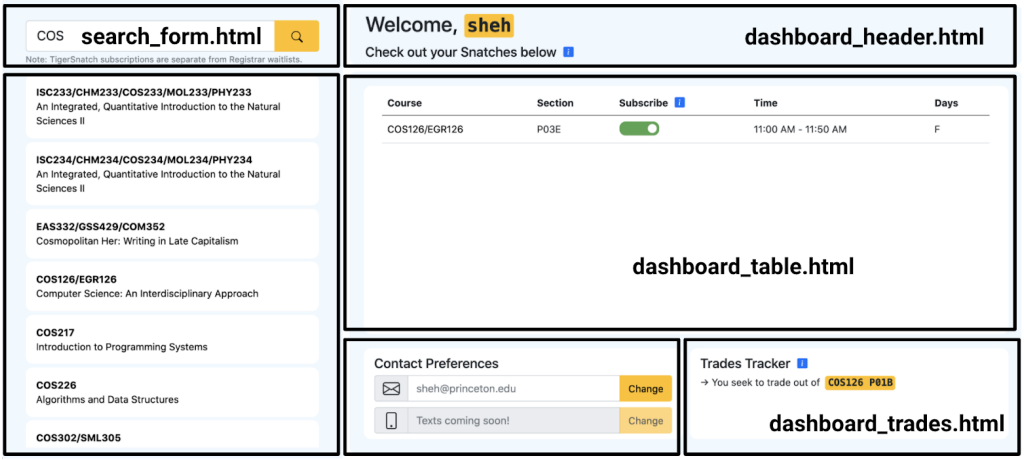
User Interface Tier

Table 1: App Pages & Jinja Templates

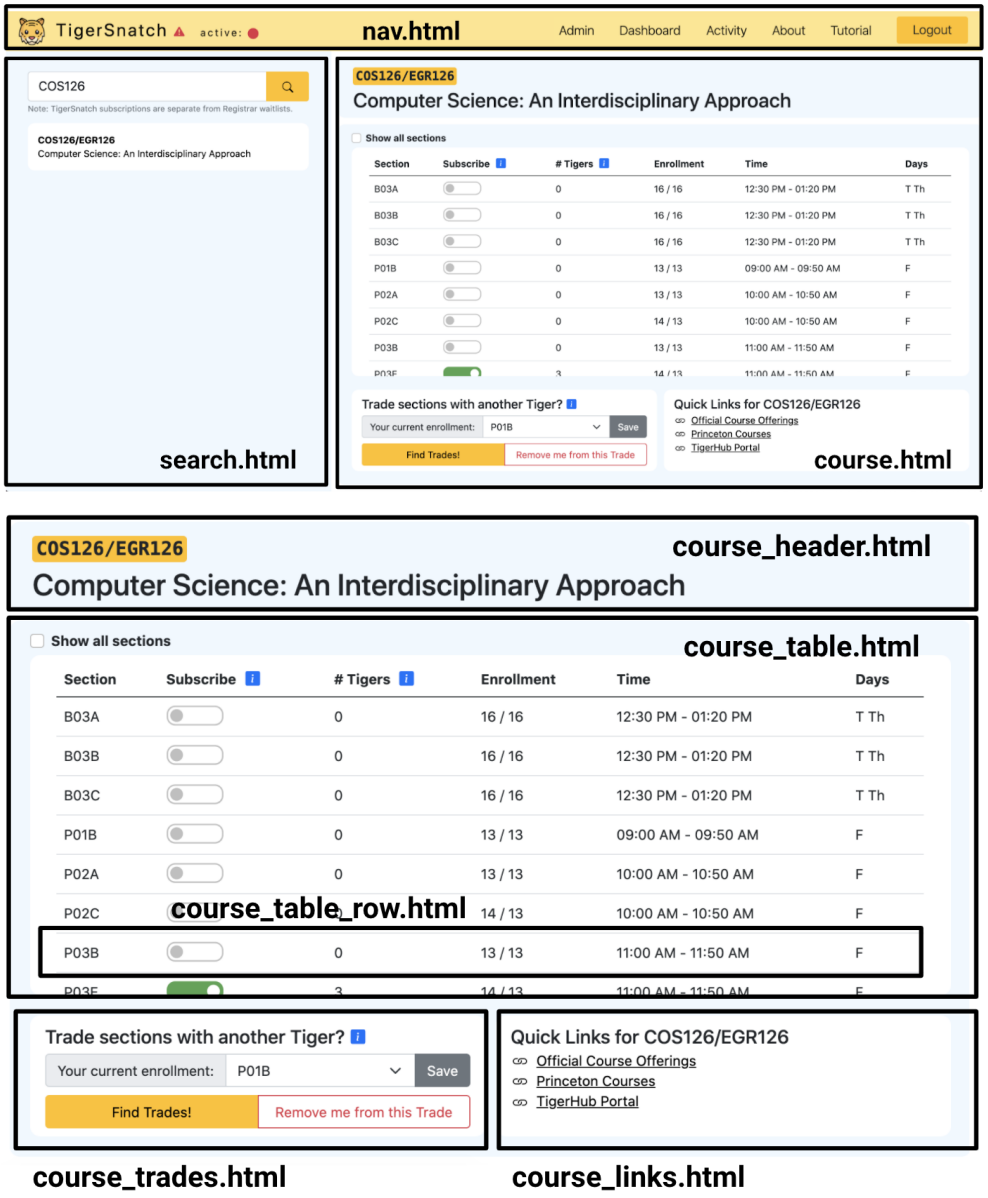
Page	Jinja Templates Used
Landing	
Tutorial	 If a user is not logged in, `nav_tutorial.html` is used as the nav bar template.

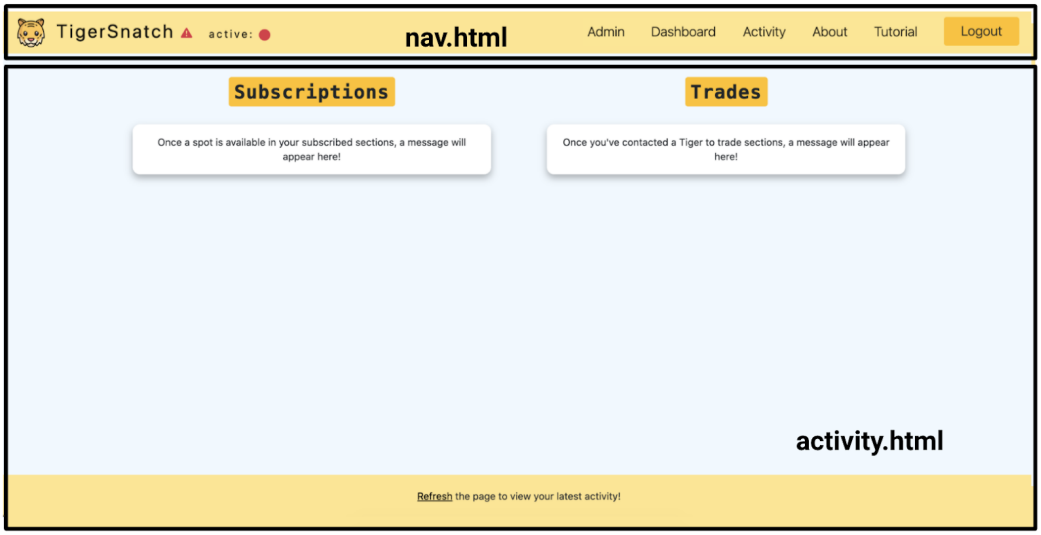
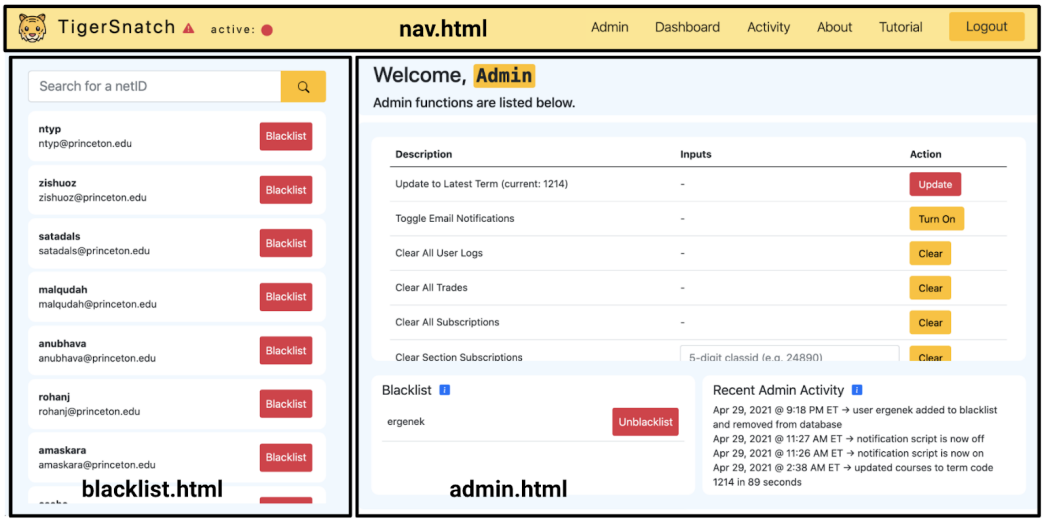
<table>
<tr>
<td>About</td>
<td>


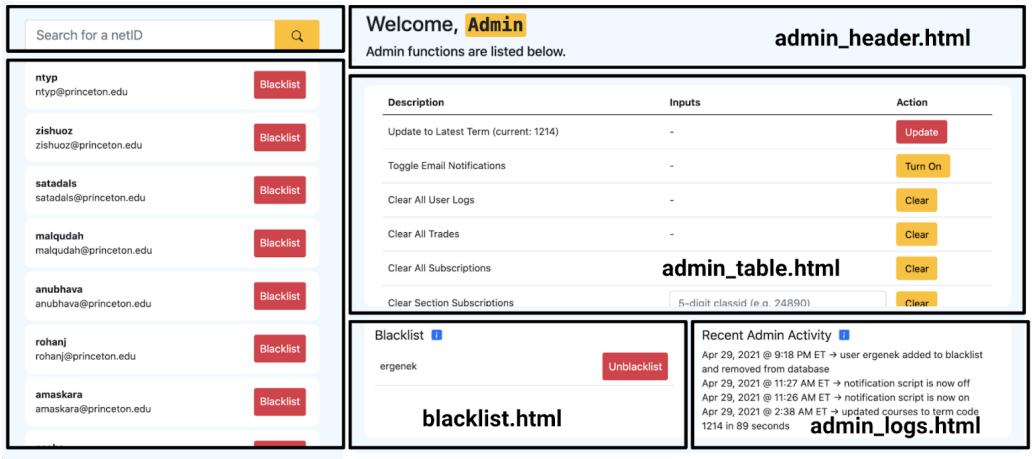
If a user is not logged in, <code>nav_about.html</code> is used as the nav bar template.
</td>
</tr>
<tr>
<td>Dashboard</td>
<td>


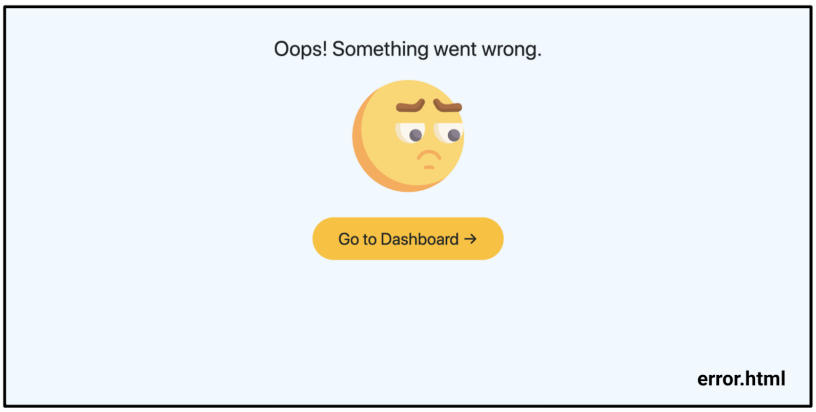
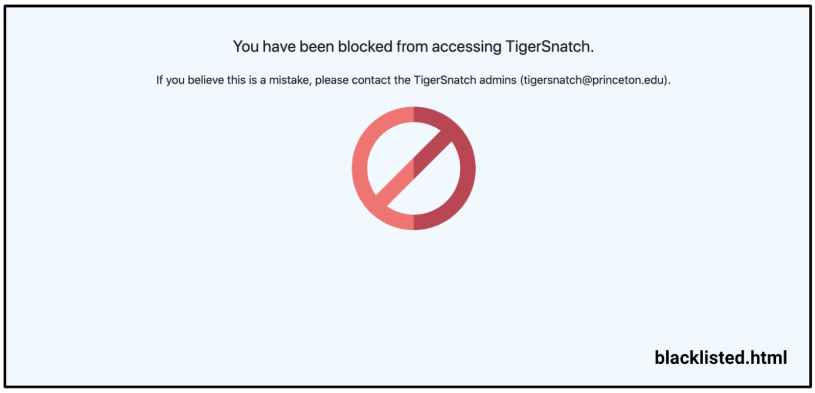
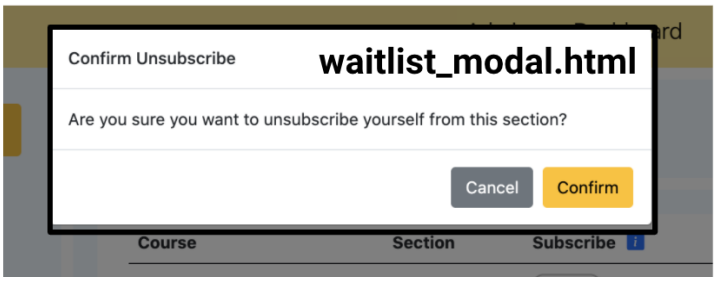
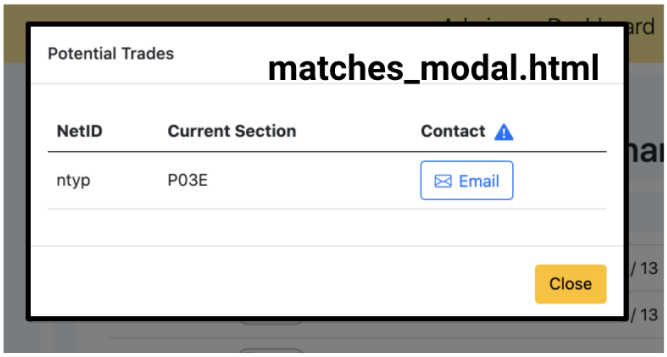


</td>
</tr>
</table>

Course	 The search templates are the same as in Dashboard.

Activity	
Admin	



Error	 
Other	 

Note that templates `common.html` and `base.html` contain HTML code that is inherited by the vast majority of our HTML templates. `common.html` handles imports for JavaScript files and

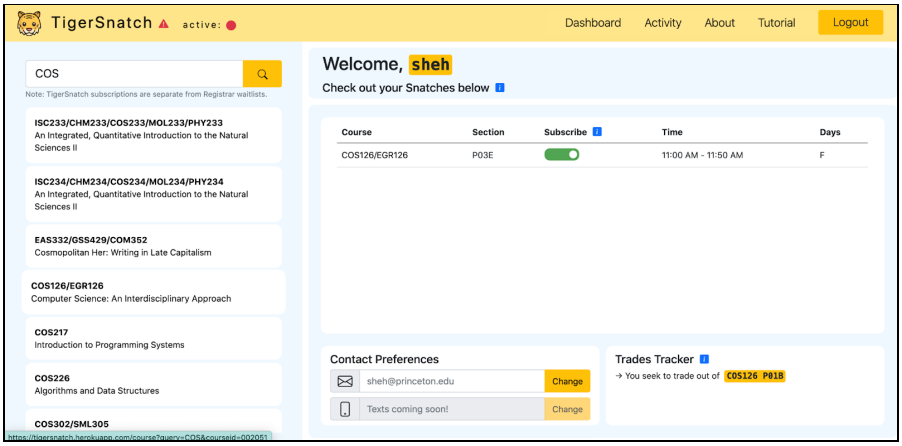
CSS stylesheets, as well as various favicon meta tags. `base.html` sets up the two-column layout for the Dashboard, Course, and Admin pages.

Data Flow Across Pages

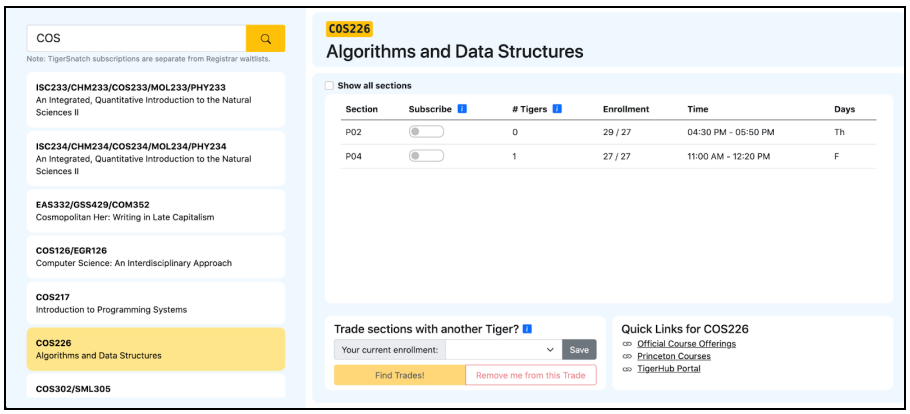
The TigerSnatch pages are largely independent, but the following are two primary instances of data flow across pages.

Dashboard to Course

On the Dashboard, the user enters a search term and clicks on a course result (e.g. COS226). The corresponding Course page is then displayed on the right, while retaining the user's search query and search results on the left.



Dashboard page

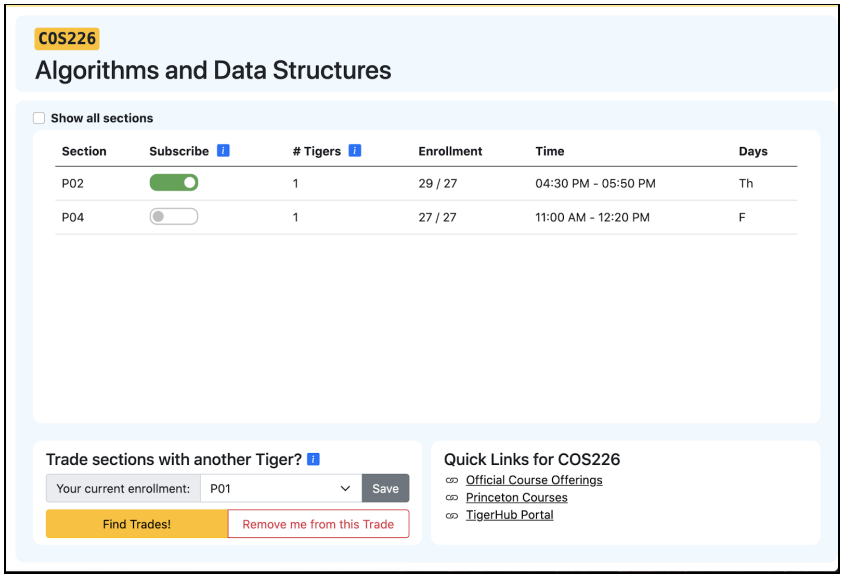


Course page

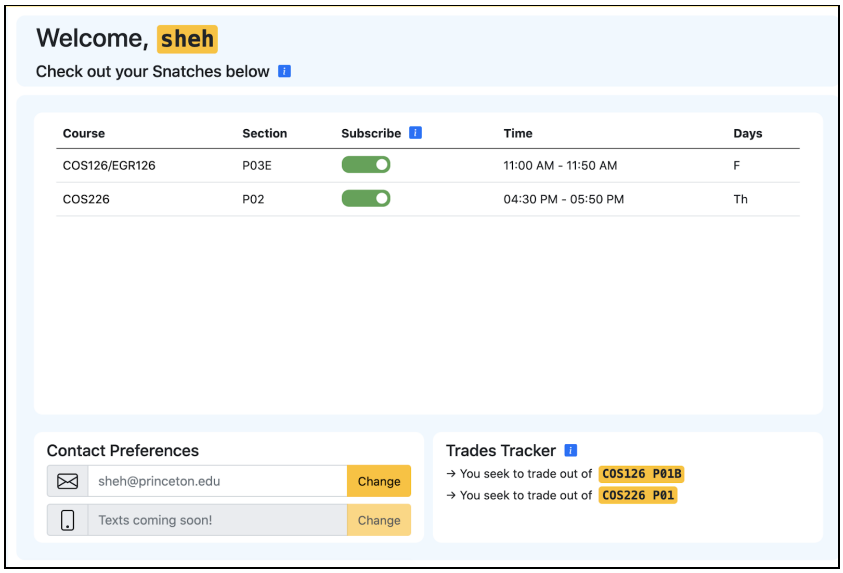
Course to Dashboard

On the Course page, the user can subscribe to sections and also save their currently enrolled section in the course (for Trades purposes). When the user returns to the Dashboard, their

Subscription is saved and viewable on the Dashboard table, while their currently enrolled section is also saved and viewable on the Trades Tracker.



Course page



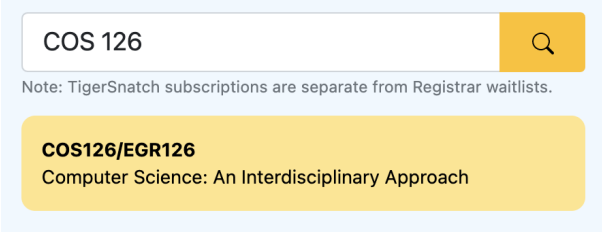
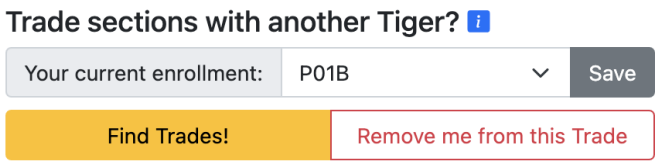
Dashboard page

Table 2: Dynamic functionality via JavaScript (defined in `app.js`)

We provide the 9 parent functions that are called when the DOM is ready. These parent functions are wrappers for additional children functions, which initialize listeners for various actions performed by a user.

Note that we use Google Closure Compiler (https://developers.google.com/closure/compiler) to generate `app.opt.js`, which is an optimized version of `app.js`. The live TigerSnatch website uses `app.opt.js`, but the table below is based on `app.js` because its code is readable and formatted. The following shell command, run inside the `/static` directory, generates `app.opt.js` from `app.js`:

```
$ closure-compiler --js app.js --js_output_file app.opt.js
```

Function Name	Description
`searchFunctions`	Listens for submission of a query in the search form or selection of a search result. 
`subscriptionFunctions`	Listens for toggling of the section Subscription switch; generates a modal asking for user confirmation when switch is toggled off. 
`tradeFunctions`	Handles all button clicks on the Trades panel of a course page that allows users to save their currently enrolled section, find Trades, and remove themselves from a Trade. 
`showAllListener`	Shows all sections of a course when the checkbox is checked.

	
`dashboardCourseSelectListener`	Listens for when a user clicks on a hyperlinked course name in the Dashboard to navigate to its course page.
`mobileViewFunctions`	Allows mobile users to scroll up and down between the course search panel and the dashboard or course details pages, and close the navbar on tap out. 
`adminFunctions`	Handles all button clicks on the Admin page, including blacklisting/unblacklisting users, updating the term, clearing subscriptions/trades, retrieving user data, and toggling email notifications.
`pageBackListener`	Listens for users to click the back button on page.
`initTooltipToasts`	Initialize all tooltips displayed on the front-end, which provide help messages when users hover over dedicated icons.

Processing Tier

Table 3: App Endpoints (defined in `app.py` and `app_helper.py`)

The following endpoints are accessible by both administrator and non-administrator users, via a URL.

Terminology

- `courseid`: the ID of a course (COS126) as taken from MobileApp API
- `classid`: the ID of a section (e.g. COS126 B03A) as taken from MobileApp API
- `classid` and `courseid` can also be found on the official Course Offerings website (https://registrar.princeton.edu/course-offerings)

Endpoint name	Request Type	Description
`/`	`GET`	Routes to `/dashboard` endpoint if user is logged in, `/landing` endpoint if not.
`/landing`	`GET`	Generates Landing page.

`/about`	`GET`	Generates About page.
`/tutorial`	`GET`	Generates Tutorial page.
`/dashboard`	`GET`	Generates Dashboard page and populates it user's data (i.e. subscribed sections, currently-enrolled sections from Trades, email). Performs course search if user query is specified.
`/course`	`GET`	Generates a specific course page populated with course information, user's Subscriptions, and user's currently-enrolled section.
`/activity`	`GET`	Generates Activity page and populates it with user's Subscription and Trade logs.
`/login`	`GET`	Implements login functionality via CAS.
`/logout`	`GET`	Implements logout functionality.

The following endpoints are accessible by both administrator and non-administrator users, not via a URL.

Endpoint name	Request Type	Description
`/dashboard`	POST	Implements functionality to allow users to update their email address.
`/searchresults`	POST	Generates course search results when no query is passed by user.
`/searchresults/<query>`	POST	Generates course search results given user search `query`.
`/courseinfo`	POST	Retrieves basic course information, along with a user's Subscriptions and currently-enrolled section for a given course.
`/add_to_waitlist/<classid>`	POST	Subscribes user to notifications for a section whose ID is `classid`.
`/remove_from_waitlist/<classid>`	POST	Unsubscribes user from notifications for a section whose ID is `classid`.
`/update_user_section/<courseid>/<classid>`	POST	Updates user's currently-enrolled section, for course `courseid`, to the given section `classid`.

`/remove_user_section/<courseid>`	POST	Removes user's currently-enrolled section (for Trades) for course `courseid`.
`/find_matches/<courseid>`	POST	Retrieves potential matches to trade sections of given course `courseid` with user.
`/contact_trade/<course_name>/<match_netid>/<section_name>`	POST	Updates user and match's (netID `match_netid`) Trade logs when user chooses to email their match for section `section_name` (e.g. B03B) in course `course_name` (e.g. COS126).

The following endpoints are accessible by only administrator users, via a URL.

Endpoint name	Request Type	Description
`/admin`	GET	Generates Admin page and populates it with admin logs, blacklisted users, and current term code. Performs search for users if a query is provided via the `query_netid` parameter.

The following endpoints are accessible by only administrator users, not via a URL.

Endpoint name	Request Type	Description
`/add_to_blacklist/<user>`	POST	Adds netID `user` to TigerSnatch blacklist.
`/remove_from_blacklist/<user>`	POST	Removes netID `user` from TigerSnatch blacklist.
`/get_notifications_status`	POST	Retrieves whether TigerSnatch email notifications are on or not.
`/set_notifications_status/<status>`	POST	Turns on or off TigerSnatch Subscriptions email notifications for all users. `status` must be either the string 'true' or 'false'.
`/update_all_courses`	POST	Updates course term and all course data (clearing all current Subscriptions and Trades).
`/clear_all_trades`	POST	Clears all current data related to Trades, including all users' current sections.
`/clear_all_user_logs`	POST	Clears Activity logs for all users.

/clear_all_waitlists	POST	Clears Subscriptions for all users.
/clear_by_class/<classid>	POST	Clears Subscriptions only for section classid.
/clear_by_course/<courseid>	POST	Clears Subscriptions for all sections of only course courseid.
/get_user_data/<netid>/<isTrade>	POST	Retrieves user's Subscriptions if isTrade is false, retrieves user's currently-enrolled sections if isTrade is true.
/fill_section/<classid>	POST	Sets enrollment equal to capacity (i.e. "fills") for section classid. Primarily for grading and testing purposes.

Table 4: TigerSnatch Core Business Logic

Module	Filename (.py)	Description
CAS	CASClient	Manages login/logout functionality with the Princeton Central Authentication Service (adapted from code by Alex Halderman, Scott Karlin, Brian Kernighan, Bob Dondero).
Subscriptions	waitlist	Contains Waitlist, a class that manages subscription-related functionality for a given user.
Notify	notify	Contains Notify, a class that sends a user email messages about open slots in their Subscribed sections.
Monitor	monitor	Contains Monitor, a class that manages enrollment updates by cross-referencing MobileApp and the TigerSnatch database. The class also manages regular updates to single courses based on user usage.
	monitor_utils	Contains utilities methods for the Monitor class.
	coursewrapper	Contains CourseWrapper, a helper class for Monitor primarily used to compute available slots for all classes in a given course.

Notifications Cron Job	send_notifs_cron	Manages regular execution (does *not* implement notifications; see `send_notifs.py` for implementation) of email notification logic using a `BlockingScheduler` wrapper. Disable/enable this cron job using the website admin panel or `_set_notif_status.py`. Execution interval `NOTIFS_INTERVAL_SECS` specified in `config.py`.
	_set_notif_status	Simple utility script to enable or disable the cron notification script on Heroku. Specify one of the following flags: `--on`: Enables the cron notification script `--off`: Disables the cron notification script Example: `$ python _set_notif_status.py --on`
	send_notifs	Script that wraps/implements core email notification logic in a single function - designed to be run on a regular interval. Recommended execution frequency: 2-5 minutes after the previous execution completion. The algorithm itself can take a minute or two to run, depending on the number of Subscribed sections.

Table 4a: Utilities for TigerSnatch Core Business Logic

Module	Filename (`.py`)	Description
N/A	_exec_server	Manages Flask server execution with a given port. To alter the host address, change TS_HOST in config.py. Takes one argument, a port number. Example: `$ python _exec_server.py 12345`
	_set_maintenance_mode	Simple script to enable or disable Heroku maintenance mode. This action can also be done from the admin web panel. Specify one of the following flags: `--on`: Enables maintenance mode

		`--off`: Disables maintenance mode Example: `$ python _set_maintenance_mode.py --on`

	`config`	The following variables in `config.py` are applicable to files in the processing tier. See the comment in the code file above each variable for more details.
		
		`TS_HOST`
		`CONSUMER_KEY`
		`CONSUMER_SECRET`
		`APP_SECRET_KEY`
		`HEROKU_API_KEY`
		`TS_EMAIL`
		`TS_PASSWORD`
		`COURSE_UPDATE_INTERVAL_MINS`
		`NOTIFS_INTERVAL_SECS`

Data Management Tier

Table 5a: TigerSnatch Core Data Management Logic

Note that the Database module interfaces with MongoDB Atlas, and the MobileApp module interfaces with Princeton OIT's MobileApp API.

Module	Filename (`.py`)	Description
`Database`	`database`	Contains `Database`, a class used to communicate with the TigerSnatch MongoDB Atlas database. Implements core database-level functions for Trades, Subscriptions, admin panel, blacklists, user management, term updates, courses, classes/sections, database resetting, and various other utilities.
	`schema`	Contains tuples of keys that various database documents must contain. Mainly used for database integrity validation.
	`_exec_reset_db`	Simple script to either soft or hard reset the TigerSnatch database. Specify one of the following flags:

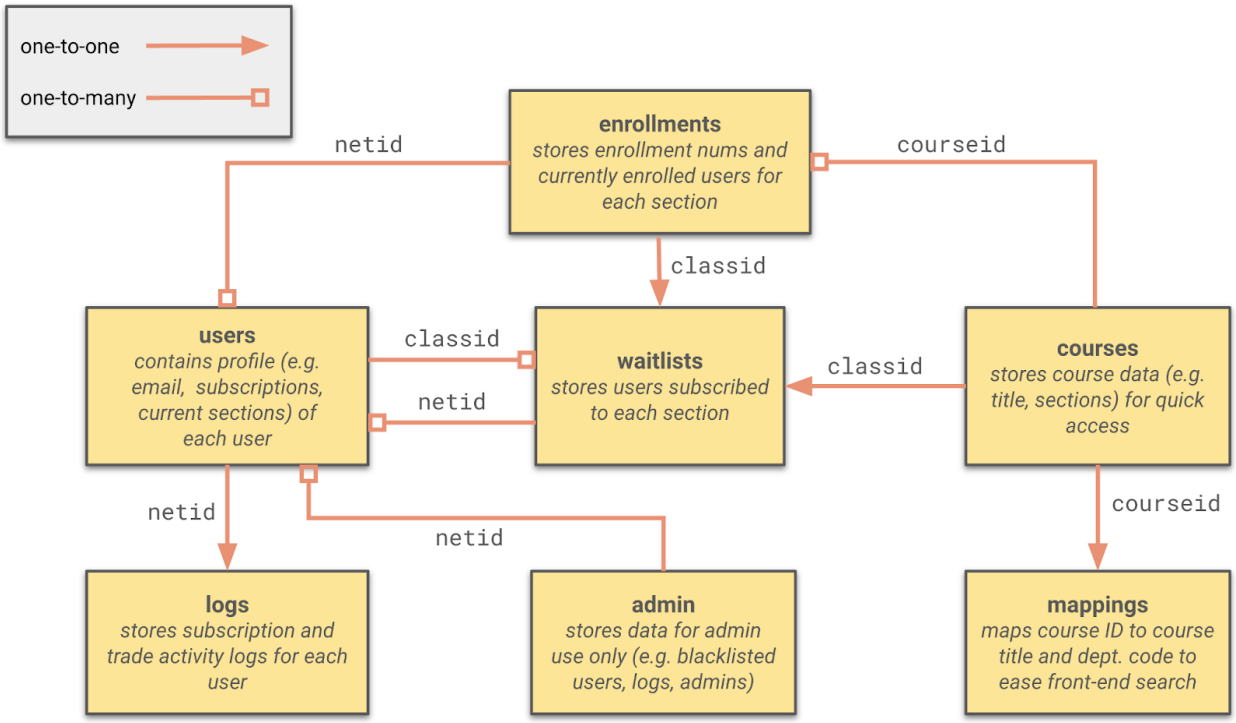
		`--soft`: resets only course-related data `--hard`: resets all non-user-profile data, including user logs, Trades, and Subscriptions **In the vast majority of cases, this script should not be executed.** Only in extreme scenarios (e.g. a major corruption) should it be run. Example: `$ python _exec_reset_db.py --soft`
	`_exec_update_all_courses`	Resets and updates the TigerSnatch database with courses from the latest term. Specify one of the following flags: `--soft`: resets only course-related data `--hard`: resets all non-user-profile data, including user logs, Trades, and Subscriptions Approximate execution frequency: once at the start of every course selection period i.e. on or after (asap) the date when courses for the next semester are released on the Registrar's Course Offerings website. **In the vast majority of cases, this script should not be executed.** Only in extreme scenarios (e.g. loss of access to the admin panel) should it be run. **This script should instead be indirectly called from the admin web panel.** See Use Case 13 in the User's Guide for more information. Example: `$ python _exec_update_all_courses.py --soft`
`MobileApp`	`mobileapp`	Contains `MobileApp`, a class used to communicate with Princeton OIT's MobileApp API. Adapted from code by `vr2amesh`.

Table 5b: Utilities for TigerSnatch Core Data Management Logic

Module	Filename (`.py`)	Description
`N/A`	`config`	The following variables in `config.py` are

		applicable to files in the data management tier. See the comment in the code file above each variable for more details.
		
		`DB_CONNECTION_STR`
		`COLLECTIONS`
		`MAX_WAITLIST_SIZE`
		`MAX_LOG_LENGTH`
		`MAX_ADMIN_LOG_LENGTH`

Database Schema



Database relationships between collections

Examples

- An example of a one-to-one relationship is that between the users and logs collection, specified via the `netid` key. Each document (i.e. row) in the users collection (i.e. table) stores user data for a unique user (netid) of TigerSnatch. Each document in the logs collection also belongs to a unique user (netid) of TigerSnatch and stores strings of their activity logs.
- An example of a one-to-many relationship is that between the users and waitlists collection, specified via the `netid` key. Each document (i.e. row) in the waitlists collection (i.e. table) corresponds to a particular course section. Each document in the

waitlists collection stores an array of `netids`, and each `netid` in the array corresponds to a unique user in the users collection that is subscribed to that particular section.

Comments

- User-specific data are contained in the following collections:
  - users, logs
- Course and section-specific data are contained in the following collections:
  - enrollments, waitlists, courses, mappings
- There exists data redundancy within the TigerSnatch database. The purpose of this is to facilitate quicker lookups.
- There also exists a collection named `system`, without relationships to any other collection, that logs important events along with timestamps (feature requested by Princeton OIT lead staff for TigerHub/PeopleSoft):
  - User login
  - User visit to a course page (e.g. COS126)
  - MobileApp query (including endpoint, arguments, response time)
  - Admin visit to admin page
  - Site maintenance mode turned on or off
  - Notifications cron job turned on or off
  - Notifications cron job executing
  - Notifications cron job summary of results (i.e. number of emails sent)

Database key-value pairs within each collection

Design Problems and Solutions

Design Problem 1: Minimizing the number of calls to Princeton OIT MobileApp API
The TigerSnatch system relies on Princeton OIT's MobileApp API for all course data, including current enrollments. One of the first design problems we faced was optimizing system efficiency with regard to the number of MobileApp API calls (obviously, fewer API calls is better). Each API call takes approximately 0.01 to 3 seconds, depending on the current load on Princeton's PeopleSoft database and on the API itself. TigerSnatch makes calls to the API during each of these three distinct events:

1. Updating the entire TigerSnatch to reflect courses for the most recent term returned by MobileApp
2. Updating all data pertaining to exactly one course (e.g. COS126)
3. Checking enrollment data for open spots in any section to which TigerSnatch users are Subscribed

Case 1 is triggered whenever an administrator activates "Update to Latest Term" on the Admin web-facing panel, or when `src/_exec_update_all_courses.py` is executed from the shell. The algorithm performs a constant number C of MobileApp queries, where C equals the number of department codes (ORF, AAS, COS, etc.). At the time of writing (Spring '21 course term), C is 103 but can vary slightly from term to term depending on course offerings. Because this use scenario occurs only once per semester and makes exactly C API calls (note that fewer than C API calls would not be possible since MobileApp was designed to query courses by exactly one department code), we were not concerned with further optimizing API usage for Case 1.

Case 2 is triggered for a particular course (assume COS126, without loss of generality) whenever both of the following conditions are met:

- A user visits the TigerSnatch COS126 page
- It has been at least 5 minutes (configurable in `src/config.py`) since the last time TigerSnatch has queried MobileApp for COS126

As a team, we had a lengthy discussion before eventually settling on the above conditions as the trigger to pull (potentially) new COS126 course data. In the initial design of TigerSnatch course-specific pages, we did not consider the notion that data other than enrollment and capacity could change. For example, the COS126 instructor could add additional precepts, or change the meeting times of labs, or even modify the course title. We quickly realized that it was absolutely necessary, in order to prevent data staleness, to poll MobileApp regularly to check for

course updates. But how often should polling happen? Should TigerSnatch poll all courses continually? Or just some courses?

For the Spring 2021 term, the TigerSnatch database contained 1,320 unique courses. We knew that it would be highly impractical and inefficient to make 1,320 MobileApp queries continually (i.e. repeatedly cycle through each course code), not to mention that our API subscription would likely be revoked or throttled by OIT. As mentioned in Case 1, an alternative solution would be to make exactly C MobileApp queries continually (i.e. repeatedly cycle through each department code). By doing this, we would reduce the number of API calls required to poll new data for all courses by a factor of about 1,320/C ≈ 13. Although these may sound like a viable solution, they in fact would result in wasted resources and compute power for the TigerSnatch backend; why poll for a course update when users are not actually viewing that course? In fact, most courses and sections at Princeton do not reach capacity, according to our internal analysis on TigerSnatch data. So, users are unlikely to search for and view those courses, thus making MobileApp polls for those courses unnecessary.

We therefore discussed options that did not involve continually polling for all courses (either via individual course codes or via department codes) and found common ground on the notion that TigerSnatch should perform queries for *only* courses that are actually being viewed by users. The only remaining question was deciding how often those queries should happen. Again, let's consider COS126. Obviously, it would be bad if we performed a query each time a user viewed COS126, since we could potentially be making dozens of identical queries in a short time frame that would result in likely only one - if not zero - actual updates to COS126 in the TigerSnatch database. We therefore decided that we should limit the rate of queries for COS126 to once per set time period, which simultaneously solves the following (very undesirable) problems:

- Continually making MobileApp queries and processing data for courses that users are not actually viewing
- Making duplicate MobileApp queries in a short amount of time for a particular course
- Having data staleness for any course

To recap, for any given course, we poll MobileApp if and only if a user visits that course page *and* it has been at least 5 minutes (or some other reasonable length of time) since the last MobileApp poll. It was fairly straightforward to implement this logic. The only side effect of Case 2 being triggered is a short delay (0.5 to 3 seconds) for users if they happen to cause a MobileApp query by viewing a course. In the worst case scenario, of course, this design decision would reduce to the undesirable situation where the system essentially polls for all courses on a continuous basis (i.e. if all courses were accessed at a 5 minute interval). However, this worst case scenario would be highly unlikely, since as stated before, only a small fraction of Princeton courses and sections actually fill up and thus most TigerSnatch course pages would never be visited.

Case 3 is triggered by the TigerSnatch email notification cron job, which runs on a regular basis during course enrollment and add/drop periods. Currently, it is configured to execute every 2 minutes, during which it will check for open slots in all sections to which at least one TigerSnatch user is Subscribed. The script's usage of MobileApp is perhaps the most intensive of the three scenarios, since it is *guaranteed* to contact the API at a set interval, provided that at least one user is Subscribed to at least one section. It was essential that we implement logic that minimizes as much as possible the number of queries we make per 2 minutes, in order to increase the speed at which users get notified about open slots and to lessen the load on PeopleSoft/TigerHub servers during critical course enrollment periods.

From the very start of developing the email notification algorithm, we knew that MobileApp has the (somewhat inconvenient) constraint that at most one course at a time can be queried, i.e. we cannot combine multiple courseIDs into a single request argument string. We have discussed this constraint with the OIT staff that maintain PeopleSoft/TigerHub and the MobileApp API, and they told us that they would consider implementing a multi-course query feature. Once they do that, we'll be able to reduce the number of queries to the API to *exactly 1 per 2 minutes*.

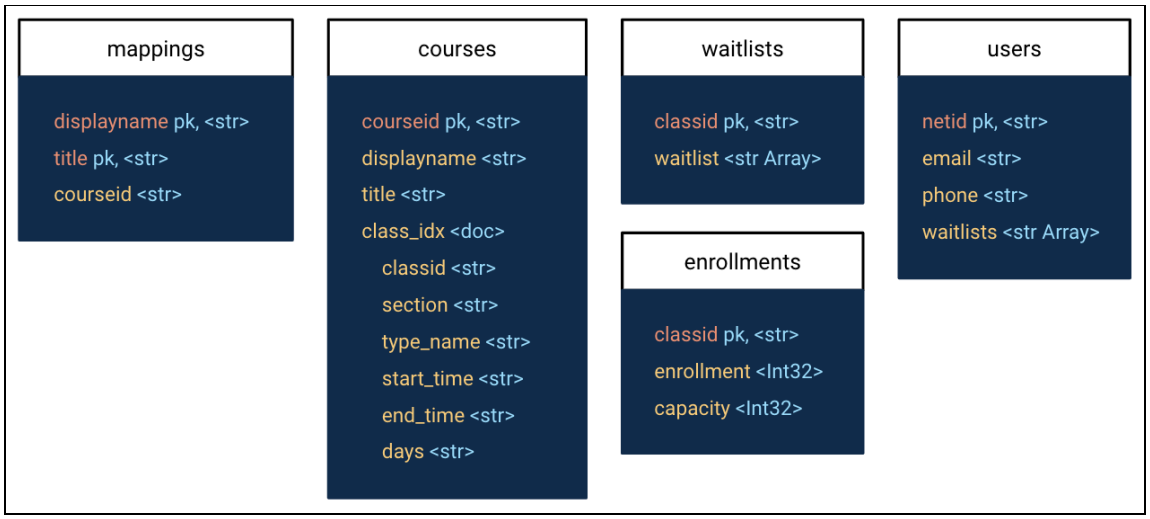
But for now, in order to minimize calls, we perform exactly 1 query per unique course every 2 minutes. If in total 49 users are Subscribed to 41 sections across 28 courses, for example, the email notification algorithm will perform exactly 28 queries to MobileApp. This is the current minimum number of calls we can possibly make. We accomplish this by first gathering all unique classIDs to which users are Subscribed, and then grouping them together based on their corresponding courseID. Note that each classID is associated with exactly 1 courseID (e.g. multiple precepts in a single course). We then convert each courseID into a valid MobileApp search string before performing a single query for all sections of that course. We then process the returned data and send out emails.

It is worth noting that the email notification script makes use of the `multiprocess` library in Python (in particular, the `Pool` class), spawning as many child processes as the system's CPU count. Each process makes exactly 1 API query and sends out emails to any user(s) Subscribed to one or more sections within the query's corresponding courseID. Using multiprocessing speeds up running time by approximately a factor of R, where R is the number of CPUs; minimizing raw running time is essential as users are depending on TigerSnatch to tell them when spots open up in their desired courses/sections.

Design Problem 2: Designing our database to facilitate quick lookups, make efficient comparisons, and adapt to the constraints of the MobileApp API

Some terminology clarification: “collections” in MongoDB are the equivalent of “tables” in SQL. Each collection contains multiple “documents” -- “documents” are the equivalent of “rows” of a table in SQL. Each document is structured like a JSON object (key-value pairs).

One design question that we faced throughout this project was how we should structure our database collections and its fields to efficiently handle field lookups. In our second week of the project, we constructed an initial database schema shown below. We first sketched out the courses collection; we consciously set up the document hierarchy and chose field names to closely mirror each course dictionary returned by the MobileApp API, in order to facilitate easy updates of our course documents later on. We also decided early on to separate out the waitlists and enrollments collection from the courses collection, despite the redundant data between waitlists/enrollments and courses. Since we frequently need to check the “`waitlist`” (i.e. list of subscribed users) for each section and enrollment data for each section, we constructed separate waitlists and enrollments collection so we wouldn’t have to traverse down a large document in the courses collection to access the same data. Likewise, since users will often enter queries to search for courses, we created a separate mappings collection that stores each course’s display name and title, so that we can easily perform substring matches between the user’s queries and these fields without having to load a large course document.

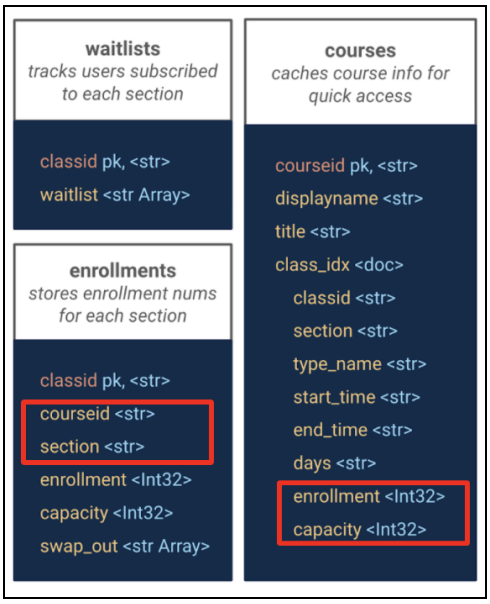


Initial database schema

As the semester progressed, we encountered additional issues that required changes to our initial database design. For instance, when a course page is opened by a user, we need to update the page with the newest course data. To do this, we must query the corresponding course dictionary from the MobileApp API and check for any differences with our cached version in the courses collection. The course dictionary (which is constructed from MobileApp data) originally did not contain enrollment and capacity fields, so that we could not simply check for equality between this new dictionary and our cached version to determine updates (i.e. capacity and enrollment changes would have overlooked) -- instead, we would have to access

fields in *both* the enrollments and courses collections to check for updates to a single course. In order to speed up comparison, we added the enrollment and capacity fields to the courses collection, as shown in our updated database schema. This way, we could simply use the equality operator to compare the MobileApp-constructed and cached dictionaries with minimal database operations.

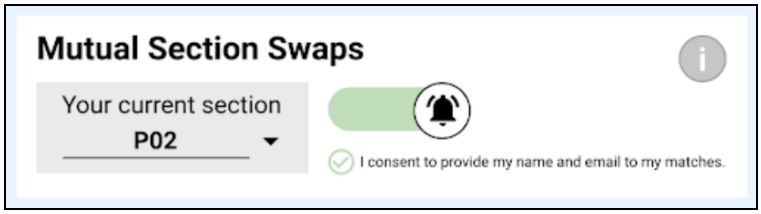
We also had to make changes in order to adapt to the constraints of the MobileApp API. In the MobileApp API, one can only query by course (not by section). So, when we check MobileApp for the latest enrollment numbers for subscribed sections, we must map the section's classID to its corresponding courseID and use this courseID to facilitate search in MobileApp. We added the `courseid` key to each section-specific document in the enrollments collection for this reason. Otherwise, we would have to iterate through each document in the courses collection and check whether the given classid is contained in that course. Likewise, we added a `section` field to the enrollments collection, so that given the classID of a user's section, we can return its section name (e.g. "P01") without having to traverse through a document in courses looking for its `section` field.



Updated database schema for enrollments & courses (changes in red)

Design Problem 3: Mutual Section Swap

One of our original core features, Mutual Section Swap (MSS), failed due to design challenges and a lack of user interest. In short, MSS is just another version of the current Trade feature, except users don't need to initiate contact with each other. Here is a sketch of what MSS would have looked like:



To illustrate more concretely, say that Person A is currently enrolled in precept P01 and subscribes to P02. Another person, Person B, indicates that they are currently enrolled in P02 and subscribes to P01. By toggling the switch, MSS automatically detects this pairing, and sends an email to both person A and person B, notifying them that there is a potential match. However, after long discussions and implementation attempts, we decided against integrating MSS into TigerSnatch due to the following reasons:

1. *Fairness:* At the beginning of the semester, we implemented Subscriptions as ordered waitlists. When a spot opens in a section, those who Subscribed to that section first will get emails from TigerSnatch first. Therefore, MSS just needed to match the first users who are registered in the MSS system, and there are not any fairness issues with who would be matched first. However, since we later decided to switch to unordered lists due to user input, MSS must incorporate a fair method with matching users.
2. *Graph Approach:* To implement MSS, we needed to maintain a large graph storing relations between all existing classes, with nodes being the classIDs, and labelled edges v → w (label: [netid list]) means [netid list] wants to swap from classid v (current section) into classid w. To detect match availability, we would look for cycles of length 2, and the edges v → w are removed once no one wants to swap from class v to class w. Not only would this graph have been large, but also it would have been inefficient to perform cycle detection, since we would be checking for MSS matches every two minutes alongside the Subscriptions notification script.
3. *User Experience:* With MSS, both matched users would receive an email (containing both netIDs) from TigerSnatch saying that they have been paired. One user would then have to send a *second* email to the other user about setting up a swap time, confirming enrollments, etc. and by the time that second email gets sent, the match may no longer be feasible. In other words, with MSS, TigerSnatch delegates more work to the matched users and wastes time, both of which detract from the user experience.

Therefore, we decided to make some changes to MSS, while keeping most of its intentions and functionalities intact, and that is how we came up with the Trade system. Now, instead of TigerSnatch finding a pair of matches, the trade system displays all potential matches, and the users must initialize all trades by sending their match an email by themselves. The new system resolves the three problems above in the following ways:

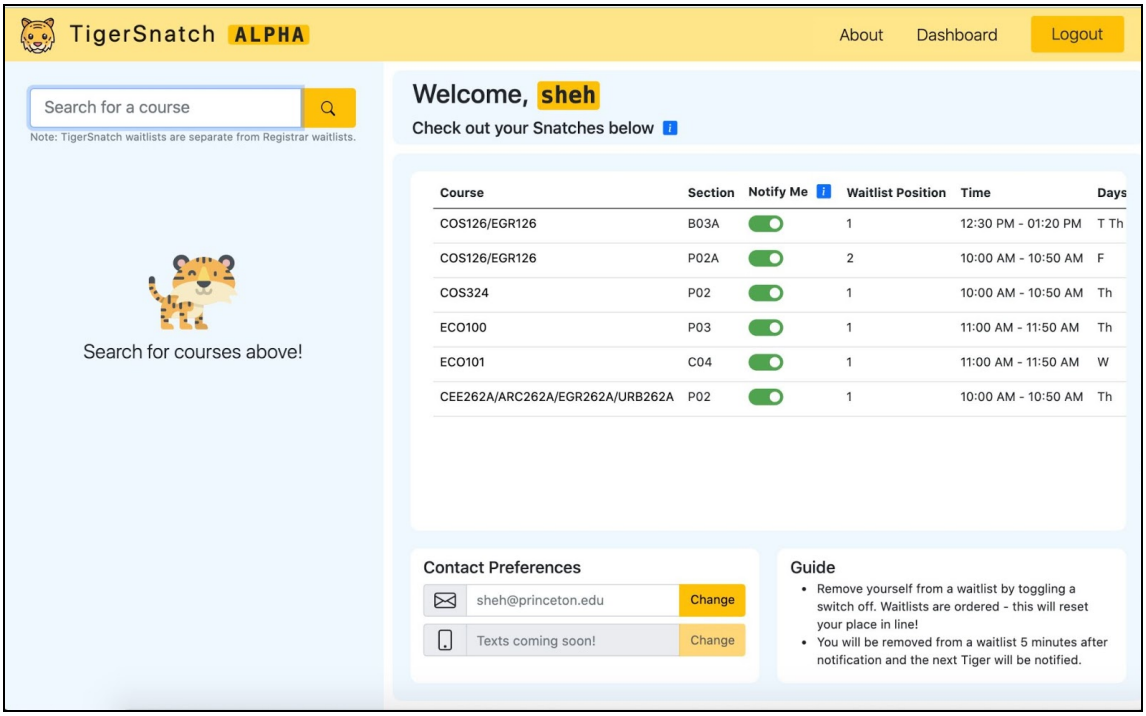
1. Since all possible matches are displayed to the user and Trade emails are sent by users themselves to matches they are interested in, we don't need to create an algorithm to

maintain the fairness of match generations. This way, we also provide users with more autonomy and flexibility. For example, the trade system avoids the situation in which a user is not on good terms with their match assigned by MSS, since the trade system prompts users with netIDs and gives them a choice for who to contact.

2. Now, we don't need to implement a graph object and perform edge detections on the entire graph. We simply needed to add a field named `swap_out` in the enrollments collection, which stores a list of netIDs of users who are currently enrolled in a class but want to swap out of it, as well as a field named `curr_sections` that stores a user's currently enrolled sections in the users collection. To get available matches for a user, we simply check whether each netID in the `swap_out` arrays for the user's Subscribed sections is subscribed to the user's currently enrolled section (see the "How Trades Work" slide in the TigerSnatch presentation for clarification).
3. Users can now select from a list of Trades matches and email them directly, instead of having to wait for TigerSnatch to notify them about their matches. In essence, the number of emails required to actually perform the Trade is reduced, improving user experience. Additionally, since users are given the responsibility to choose their match and click send on an email to a match, they are held accountable for fulfilling their Trades, and malicious users are less likely to engage in false Trades.

Design Problem 4: Enhancing user onboarding experience

In our original design of the app, we had large chunks of instructions that took up a decent portion of the Dashboard and Course page in a section called "Guides", like so:



We originally included the Guide sections so novice users could reference them to understand how features worked. However, not only are those instructions very short and somewhat unclear, but also our alpha testers reported that they did not read over the Guides when they used our site. In addition, we had ideas for implementing additional features, and due to the presence of the Guides, we could not fit all this content onto one page. Therefore, we decided to gather all information that can be helpful to users onto a separate “Tutorial” page. The new Tutorial page provides step-by-step instructions on how users can navigate through TigerSnatch. It is shown to users right after they log in for the first time, and remains accessible to all users via a navigation bar link. By factoring out these instructions into a separate page, we minimized the amount of text on the Dashboard and Course pages and made space for more helpful features. For instance, the Guide on the Dashboard was replaced with the Trades Tracker panel, which displays all classes that a user seeks to Trade out of; and the Guide on Course pages was replaced with the Quick Links panel, which provides useful links (Registrar’s Course Offerings page, Princeton Courses, TigerHub portal) that contain additional details not displayed on TigerSnatch about a specific course.