Semester.ly Documentation

Release 1.0

Semester.ly Technologies, LLC

Feb 02, 2019

Contents

Note: Want your school on Semester.ly? We want it too! Want to see a new feature to help your peers? Let's make it happen. We want to help you make the impact you want to see. We'll even find you something impactful to work on if you're not sure where to start.

Built for students by students. Semester.ly is a web platform created to bring modern technology to the most difficult and archaic parts of higher education. It all started with one problem, universal across all college campuses: course registration is a pain. Spreadsheets, sticky notes, PDFs of course evaluations, and an outdated registration platform....it is all too much in the heat of classes and exams. We set out with the mission to make college more collaborative and more stress free.

CHAPTER 1

What We Believe In

Today, we work to solve many more exciting problems in this space across many more universities. However, our fundamental beliefs remain the same:

Course registration should be easy

> Picking the right classes should be quick and painless. We believe high quality, centralized, and shareable information makes for better decision making. By doing the legwork for you, Semester.ly gives you more time to study for your courses, and decreases the time spent studying which classes to take.

Education should be collaborative

> Studies show the positive impact that friendship has in higher education classrooms. Having courses with friends and a tigther knit university community increases student success and retention. That's why Semester.ly helps students find courses with friends and helps new students make new friends in their classes.

Students know best

> Universities can't keep up with technology. Most university systems aren't even mobile responsive! Forget about using social media. That's why Semester.ly is built by students, and always will be. That's why we are open source. Oh, and it's why we use emojis .

Installation

This guide will bring you through the steps of creating a local Semester.ly server and development environment. It will walk through the setup of the core ecosystems we work within: Django/Python and React/Node/JS. It will additionally require the setup of a PostgreSQL database.

Fork/Clone The Repository

Forking Semester.ly will create your own version of Semester.ly listed on your GitHub! Cloning your Semester.ly fork will create a directory with all of the code required to run your own local development server. Navigate to the directory you wish to work from, then execute:

1. **Fork** navigate to our GitHub repository then, in the top-right corner of the page, click Fork.
2. **Clone** by executing this line on the command line:

   Note: ATTENTION: Be sure to replace [YOUR-USERNAME] with your own git username

   ```
   git clone https://github.com/[YOUR-USERNAME]/semesterly
   ```

Option 1: Set up using Docker

Steps are below on getting your local development environment running:

1. **Download and install docker** for your environment (Windows/Mac/Linux are supporter)

   https://www.docker.com/get-started

2. Create **semesterly/local_settings.py** as follows:

   ```
   DEBUG = True

   TEMPLATE_DEBUG = DEBUG

   DATABASES = {
       'default': {
           'ENGINE': 'django.db.backends.postgresql_psycopg2',
           'NAME': 'postgres',
           'USER': 'postgres',
           'PASSWORD': '',
           'HOST': 'db',
           'PORT': '5432',
       }
   }
   ```

   Note: ATTENTION: When you clone the repo, you get a folder called semesterly and inside there is another folder called semesterly. Put this in the second semesterly folder.

3. **Edit semesterly/dev_credentials.py and add a value for JHU_API_KEY in single quotes like below.** You can request this API KEY from http://sis.jhu.edu/api.

   ```
   'JHU_API_KEY': 'xxxxxxxx',
   ```

   Note: ATTENTION: This is also in the second semesterly directory.

Now run this command in your terminal to make sure that this file isn't tracked by Git and your API key stays local to you.

```
git update-index --skip-worktree semesterly/dev_credentials.py
```

3. Add this entry to your hosts file as follows (This file is in c:WindowsSystem32driversetchosts or /etc/hosts)

   ```
   127.0.0.1       sem.ly jhu.sem.ly
   ```

   Note: **ATTENTION:** If you're working on other schools, add their URLs here as well (i.e. uoft.sem.ly for University of Toronto).

4. Launch terminal or a command window and run:

   ```
   docker-compose build

   docker-compose up
   ```

   The **build** command creates a local Database and build of your source code. The **up** command runs everything. Be careful not to build when you don't need to as this will destroy your entire database and you'll need to ingest/digest again to get your course data (which takes about 30 minutes).

   You now have Semester.ly running. If this is the first time, you will want some data which done in the next step.

5. Getting JHU data for a given term. In a new terminal run the following

   ```
   docker exec -it $(docker ps -q -f ancestor=semesterly) /bin/bash
   * OR if that doesn't work
   docker exec -it $(docker ps -q -f ancestor=semesterly) shell
   ```

   This will put you inside of the shell. Now you can get courses by running these commands:

   ```
   python manage.py ingest jhu --term Spring --years 2018

   python manage.py digest jhu
   ```

6. Open a browser and visit http://jhu.sem.ly:8000 and hack away.

   You can skip ahead to **Advanced Configuration** or **How it All Works** now.

Option 2: Setup using a Python Virtual Environment

Make sure you have installed Python 2.7. If you have not you can follow this. Please also download the python installer, PIP (install guide). We will now install and setup a python virtual environment. This keeps your dependencies for other projects and classes seperate from those required for Semester.ly.

Install virtualenv:

```
sudo pip install virtualenv
```

Create a virtual environment called `venv`:

```
virtualenv -p /usr/bin/python2.7 venv
```

To enter your virtual environment, execute the following code from your Semesterly directory:

```
source venv/bin/activate
```

Note: Be sure to execute the above "source" command anytime you are working on Semesterly!

Check your OS info

If you're on a posix OS (Mac, Ubuntu, Fedora, CentOS, etc.) this is how you check what version of OS you're on.

```
uname -n
```

Install PostgreSQL

Before installing the python requirements, you must make sure to have PostgreSQL setup on your device.

On mac, install Homebrew and run:

```
brew install postgres
pg_ctl -D /usr/local/var/postgres start && brew services start postgresql
```

On Ubuntu 14.x.x use apt-get:

```
sudo apt-get install postgresql python-psycopg2 libpq-dev libxslt-dev libxml2-dev
```

On Ubuntu 16.x.x use apt:

```
sudo apt install postgresql python-psycopg2 libpq-dev libxslt-dev libxml2-dev
```

On CentOS / Fedora use yum:

```
sudo yum install postgresql gcc python-lxml postgresql-libs libxslt-devel libxml2-
↪devel
```

Install Python Requirements

Note: **ATTENTION MAC USERS:** you must install the xcode command line tools via `xcode-select --install` before proceeding. You may also need to update openssl. If so, please follow this guide.

All python dependencies are kept in a file called `requirements.txt`. Anytime a dependency is added or changed, we update it in this file. To bring your virutal environment up to date with all of these requirements easily, simply execute:

```
pip install --upgrade pip
pip install -r requirements.txt
```

There are python modules that are missing from requirements.txt. Install them with:

```
pip install pyyaml pygments kombu==3.0.33 billiard
```

Install Node Packages

Node and node package manager are the backbone of our frontend setup. To begin, install Node Package Manager (npm).

On mac:

```
brew install node
```

On Ubuntu 14.x.x:

```
wget -qO- https://deb.nodesource.com/setup_6.x | sudo bash -
sudo apt-get install nodejs
sudo apt-get install npm
```

On Ubuntu 16.x.x:

```
wget -qO- https://deb.nodesource.com/setup_6.x | sudo bash -
sudo apt install nodejs
sudo apt install npm
```

On CentOS / Fedora:

```
sudo yum install -y gcc-c++ make
curl -sL https://rpm.nodesource.com/setup_6.x | sudo -E bash -
sudo yum install nodejs
```

Then use the newly installed Node Package Manager (npm) to install all javascript dependencies. When you execute this command, it reads from the file `package.json` which specifies all dependencies, their versions, and some additional node related configurations:

```
sudo npm install
```

Setup Your Dev Environment

Now that all of the requirements are installed, its time to get your environment up and running.

Setup Your Database

Semester.ly stores objects like courses, timetables, and students in a Postgres database. Let's get one setup for you.

Let's first initialize Postgres using the default user account `postgres`

Note: If using Linux log into this account with

```
sudo -i -u postgres
```

Then, enter Postgres environment with

```
psql postgres
```

Note: If you see an error in CentOS / Fedora, it's most likely due to postgres is not running. Initialize it with `sudo service postgresql initdb && sudo service postgresql start`.

Here you can enter SQL to create/manipulate/access databases. Let's create a Semester.ly database. Enter:

```
CREATE DATABASE semesterly;
```

Then, create a database user, set `myusername` and `mypassword` to whatever you wish

```
CREATE USER myusername WITH PASSWORD 'mypassword';
```

Finally, grant all access to the created database to your new user, `myusername`:

```
GRANT ALL PRIVILEGES ON DATABASE semesterly TO myusername;
```

Great. You are all set. Enter the following to quit psql:

```
\q
```

Note: If using Linux exit postgres by

```
exit
```

Note: For CentOS / Fedora, Change all occurances of ident to md5 in pg_hba.conf. You can modify the file through `sudo vim /var/lib/pgsql9/data/pg_hba.conf`. After you change it, restart postgres with `sudo service postgresql restart`.

Create Local Settings

Now that you have a database created we need to inform Django of the configuration. Do so by creating a new file called `local_settings.py` and placing it in the `semesterly/` directory within your workspace. You should find that there is already a similar file called `settings.py` found in the same folder.

The contents of this file should be:

```
DEBUG = True

TEMPLATE_DEBUG = DEBUG

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'semesterly',
        'USER': 'myusername',
        'PASSWORD': 'mypassword',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Note: Be sure to change the values of `myusername` and `mypassword` to the values you chose when creating your user!

Migrate Your Database

Now that Django knows about the database, it can conform the empty database to our schema. Simply execute:

```
python manage.py migrate
```

Edit your /etc/hosts

For development purposes, we map http://sem.ly:8000 to http://localhost:8000. To do this locally, execute the following line of bash:

```
sudo sh -c "echo '127.0.0.1       sem.ly jhu.sem.ly uoft.sem.ly vandy.sem.ly chapman.
↪sem.ly umich.sem.ly gw.sem.ly umd.sem.ly' >> /etc/hosts"
```

Note: If you add a school, be sure to add it to this file!

Set your Environment Type

Add the following line to either your `~/.bashrc` or `~/.zshrc` which tells webpack you are running a development environment:

```
export NODE_ENV=development
```

Then `source ~/.bashrc` or `source ~/.zshrc`

And make sure the following line returns “development”

```
echo $NODE_ENV
```

Install & Run Webpack

Webpack compiles our React componenets into one application wide javascript bundle. We use chromedriver to test them.

To install them if you are testing in chrome install:

```
npm install -g webpack chromedriver
```

To install them if you are using firefox or a 32 bit operating system (like lubuntu) run:

```
npm install -g webpack
```

Then run it with:

```
npm run watch
```

Note: Always leave `npm run watch` running. It will continuously watch your javascript files and recompile automatically upon any edits/changes.

Running the Server

Now, the moment you've all been waiting for! Let's run the server! (Be sure to leave the last `npm run watch` command running)

```
python manage.py runserver
```

Navigate to http://sem.ly:8000, and if everything loads, you should be all set :). You did it!

Your Final Setup

Great work. Your Semester.ly local environment is all setup.

Don't forget: **whenever you are working on Semester.ly** you should have one terminal running the server (via `python manage.py runserver`), and one running webpack (via `npm run watch`).

Note: Don't forget to always work from your virtual environment! From the root directory, just execute `source /venv/bin/activate` to enter it.

Happy hacking! To fill up your database, be sure to checkout *Loading the Database*.

Loading the Database

To load the database you must ingest (create the course JSON), validate (make sure the data makes sense), and digest (load the JSON into the database). You can do so using the following commands:

Ingest

Note: If you have ingested before and still have the JSON file on your device, you may skip ingesting and simply digest the old data. This is useful if you are resetting your database during development and wish to quickly reload course data.

```
python manage.py ingest [SCHOOLCODE]
```

You may leave out the school code to parse all schools. This will run for a substantial amount of time and is not recommended.

Note: To parse JHU data, you will need to acquire an API access key from SIS. Add the key to `dev_credentials.py` in the `semesterly/` directory.

Digest

```
python manage.py digest [SCHOOLCODE]
```

You may leave out the school code to digest all schools.

Learn More & Advanced Usage

There are advanced methods for using these tools. Detailed options can be viewed by running

```
python manage.py [command] --help
```

For example, you can use the term and year flags to parse only a specific term:

```
.. code-block:: bash
```

> python manage.py ingest [SCHOOLCODE] –term Fall –year 2017

If you are developing a parser or contributing to the pipeline design, you will more than likely need to learn more. Checkout *Data Pipeline Documentation* or *Add a School*

Note: This step is not neccessary for most developers. Only continue reading this section if you need to override the test secrets (API keys/credentials) provided by Semester.ly (which are for testing only).

Advanced Configuration

Semester.ly makes use of several secrets which allow it to interact securely with third party software providers. These providers include Facebook (for oauth and social graph), Google (oauth), and university APIs.

In order for Semester.ly to run out of the box, we have included credentials to test Google and Facebook applications for development purposes. We override these keys for production use thereby keeping our client secrets... well, secrets! These provided credentials can be found in `semesterly/dev_credentials.py`:

```
SECRETS = {
    #Credentials for a test application for Semester.ly (+ Google/Facebook)
    'SECRET_KEY': ...,
    'HASHING_SALT': ...,
    'GOOGLE_API_KEY': ...,
    'SOCIAL_AUTH_GOOGLE_OAUTH2_KEY': ...,
    'SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET': ...,
    'SOCIAL_AUTH_FACEBOOK_KEY': ...,
    'SOCIAL_AUTH_FACEBOOK_SECRET': ...,
    'FB_TEST_EMAIL': ...,
    'FB_TEST_PASS': ...,

    #Not essential for testing, but can be filled in for advanced usage
    ...
}
```

Overriding/Setting Secrets

However, if you wish to override these credentials or add login credentials for a school which requires a client secret, you may add your key/value pair to `semesterly/sensitive.py`. This file is gitignored and will be kept private so you can safely store the private information you wish within this file. It should have a format indentical to SECRETS above and in `semesterly/dev_credentials.py`.

Using Secrets

In order to properly access a secret from anywhere within the code, simply import the `get_secret` function and use it to access the secret by key:

```
from semesterly.settings import get_secret
hashids = Hashids(salt=get_secret('HASHING_SALT'))
```

This will check the following locations for the secret (in order, using the first value it finds), throwing an error if it does not find the key at all:

1. Check OS environment variables
2. Check `semesterly/sensitive.py`
3. Default to `semesterly/dev_credentials.py`
4. Error

How it All Works

A high level description of how Semester.ly works, and what parts do what

Semester.ly pulls data about courses, exams, ratings, and more from all across the internet. It saves this data into a custom representation within a **Postgres database**. The data is retrieved using a variety of **webscraping, HTML parsing, and information retrieval** techniques which we've built into our own mini-library of utilities. This data is entered into the database via the **Django** ORM (Object-Relational Mapping). The ORM allows us to query the database and create rows using python code as if these rows were objects.

We manipulate and access this same data using Django **views** to respond to any web requests directed to our server. For example, when a user clicks on a course to open the course modal, the browser issues a request asking for the data related to that course. Our Django views respond with a **JSON** representation of the course data for rendering on the UI.

The browser knows when and how to make these requests, as well as how to generate the UI based on the responses using React and Redux. **React and Redux** maintain application state and use **Javascript** to render **HTML** based on that state.

Finally, this HTML is styled with **SCSS** for an appealing, cohesively styled user experience!

The Apps that Make Semester.ly

The overall, the Semester.ly application is made up of many smaller *apps* which each handle some collection of logic that makes Semester.ly tick! Each app encapsulates a set of urls which map a request to a view, views which respond to requests with HTML/JSON/etc, models which represent tables in the database, and tests which ensure Functionality behaves as expected.

App Name	Key Models/Functionality	Description
Semesterly	Root app. No core models, views, or functionality.	Delegates urls to sub-apps, contains end-to-end tests, other configuration.
Timetable	**Models:** Course, Section, Offering, Timetable, Textbook, Evaluations	Timetable generation and all models required for timetable representation.
Courses	Course Serializer, Views for returning course info	Functionality for accessing course data, the course modal, course pages
Authpipe	Authentication, login, signup	Authentication pipeline functions for the authentication of users, creation of students, and loading of social data via Python Social Auth
Analytics	**Models:** SharedTimetable, DeviceCookie, Feature Views	Tracks analytics on the usage of features as objects in the database. Renders a dashboard at /analytics.
Exams	Final exam share model, views for serving final exam schedule	Contains the logic for inferring exam schedules from course schedules
Integrations	Integration views	Functionality for integrating school specific code to appear in search or in the course modal
Searches	Advanced search, basic search	Views for parsing queries and returning course data
Students	Student, Personal Timetables, Reactions, Personal Event	All logic for logged-in specific users. Creating and saving a personal timetable, reacting to courses, saving custom events.
Parsing	Scrapers, parsers, parsing utilities	Home of the data pipeline that fills our database

Learning The Stack

Note: Learning a new thing can be scary. Especially when all you have are some docs and a massive code base to learn from. That's why we are here to help you learn, build, and contribute. Ask us questions! contact@semester.ly

Our Stack

Component	Technology	Style/Methodology	Tutorials
Database	PostgreSQL	Django ORM	Making Queries with Django
Backend Framework	Django	PEP8	Writing your first Django app
Frontend Framework	React	Redux/Airbnb	React Basics, React, Redux.
CSS Framework	SCSS	BEM/Airbnb	CSS Basics, SCSS, BEM

Tutorials and Resources

Learning the Backend

Django is a Python Web framework that provides a huge number of tools for web developers to quickly write scalable code with minimal configuration. It is used all over the tech industry by companies like Spotify, Instagram, YouTube, and DropBox!

Writing your first Django app is the official Django tutorial. It is top notch! The official documentation can be found at the same url and provides high quality information about how to build with this modern web framework.

Learning React/Redux

React is a Javascript library created by Facebook for “building user interfaces”. It allows developers to make encapsulated components that can be written once and used anywhere.

Redux is state container that makes React development easier to manage long term!

If you’re a beginner, we’ve created a React tutorial that will teach you the basics of developing a React app all via codepen. This is a great way to get started!

We highly recommend continuing with React via EggHead’s React Fundamentals video which teaches you everything you’ll need to know.

Finally, finish off with EggHead’s Redux tutorial. You’ll be a pro after that!

Learning CSS/SCSS

The most important step is to learn the CSS basics.

With that, you can dive into SCSS, a css preprocesor.

For development, we use the BEM methedology (learn about BEM here!) and the Airbnb style guide.

Learning Scraping/Parsing

Our own tutorial, coming soon!

How to Contribute

Contributing to Semester.ly follows the following simple workflow:

1. *Fork the Repository*
2. *Make Changes (fix a bug, create a feature)*
3. *Open a Pull Request (and see your code go live!)*

Fork the Repository

Follow the instructions in the installation portion of the documentation, see *Installation*

Make Changes (fix a bug, create a feature)

Add the Upstream Repo

You’re going to want to add the original project repo as an upstream repo in your forked project:

```
git remote add upstream git@github.com:noahpresler/semesterly.git
```

This way you can push to your fork as “origin” and the main repo as “upstream”. You’ll only ever do this once.

Syncing With Upstream

To stay up to date with upstream/master, you'll consistently want to checkout the master branch, fetch the upstream changes. Merge these into your local master branch and push that merge. These lines do exactly that:

```
git checkout master
git fetch upstream
git merge upstream/master
git push origin master
```

Create a Working Branch

Now you'll want to checkout a branch off master to work on. This is the branch you will merge into upstream when you are done. Just do:

```
git checkout -b mybranchname
```

Make Some Changes, Add and Commit

After you've made edits, git add your files, then commit. One way to do this:

```
git commit -a
git push origin mybranchname
```

Note: **What If Upstream Has Changed?** Just pull and rebase onto those changes and push. You may find conflicts, that's to be expected!

```
git pull --rebase upstream master
git push origin mybranchname
```

Open a Pull Request (and see your code go live!)

So you've made your changes, and you've pushed them to your branch. To open a PR, simply head over to your fork at: https://github.com/YOURGITHUBUSERNAME/semesterly. Click on "Pull Request", choose the upstream repo "master" as the destination, and your forked repo's branch (the one you've been working on) as the source, and pick the merge and squash option!

Awesome! You've made a PR. Once its merged, your code will be a part of the Semester.ly open source GitHub repository and will be deployed for tens of thousands of students to use/benefit from.

Note: A PR must pass a few checks before it can be merged.

LGTM: Before your PR is merged, you'll need to pass a peer review to ensure that all the changes are clean and high quality. Usually, you'll get an "lgtm" (the comment which triggers this check to pass) or a few minor edits will be requested. This helps us maintain a quality code base and helps contrbutors learn and grow as engineers!

PR Body: Your pull request should reference a git issue if a related issue has been created. Additionally, it must provide an in depth description of why the changes were made, what they do, and how they do it. This message can be formatted as *"WHY: ...., WHAT:....., HOW:....."*, but it can take any form if this does not suit your case.

Tests & Builds Pass: All tests and builds, as run by TravisCI must pass.

Linting Satisfied: All files must successfully pass our code style checks. You can check that your code has no errors by running:

```
npm run lint
```

You can learn more about how lint checking is done by reading *Learning The Stack*.

Add a School

Adding a new school is easy and can be done in a few simple steps:

1. *Run the Scaffolder*
2. *Develop the Parser*
3. *Parse and Test*

Run the Scaffolder

Running the *makeschool* command will create a directory for your school, creating a configuration file, a stub for the parser, etc. Run the following for your school:

```
python manage.py makeschool --name "University of Toronto" --code "uoft" --regex "([A-
↪Z]{2,8}\\s\\d{3})"
```

Don't forget to add this new school to your /etc/hosts! (Check here for a reminder on how: *Setup Your Dev Environment*)

Develop the Parser

Note: Notify us if you intend to add a school! Create a GitHub issue with the tag new_school. We can help you out and lend a hand while also keeping track of who's working on what!

The scaffolder created the stub of your parser. It provides the start function and two outer loops that iterate over each provided term and year. **Your goal is to fill the inside of this so that for each year and term, you collect the course data for that term/year.**

What this boils down to is the following template:

```
for year in years:
    for term in terms:

        departments = get_departments(term, year)

        for department in departments:

            courses = get_courses(department)

            for course in courses:
                self.ingestor['course_code'] = ...
                self.ingestor['department'] = ...
                self.ingestor['description'] = ...
                ...
                self.ingestor.ingest_course()

                for section in sections:
                    self.ingestor['section_code'] = ...
                    self.ingestor['section_type'] = ...
                    self.ingestor['year'] = ...
                    self.ingestor['term'] = ...
                    ...
                    self.ingestor.ingest_section()

                    for meeting in meetings:
                        ...
                        self.ingestor.ingest_meeting()
```

Breaking it down

The code starts out by getting the departments. It doesn't have to, but often it is easiest to go department by department. The parser then collects the courses for that department. We will talk about how it does this later in *How To Fill The Ingestor*.

For each course, the parser fills the ingestor with the fields related to the course (e.g. description, the course code). Once complete, it calls *ingest_course* to execute the creation of the course.

It then repeats this process for the sections belonging to that course, and for each section, the meetings (individual meeting times) belonging to the section.

Everything else is handled by the BaseParser and the ingestor for you.

How To Fill The Ingestor

As shown by the code sample above, filling the ingestor is as easy as filling a python dictionary. The only question that remains is how to collect the data to fill it with.

The answer is by pulling it from the internet of course! Luckily we have a tool called the **Requester** which helps developers like you to *request* information from a web course catalogue or API.

Using the Requester

By inheriting from the BaseParser, your parser comes with its own requester that can be used like this:

```
markup = self.requester.get('www.siteorapi.com')
```

or:

```
markup = self.requester.post('www.siteorapi.com', data=form)
```

It will automatically return a markedup version of the data returned by the request (automatically detecting JSON/XML/HTML).

Note: The requester will maintain a session for you, making sure the proper cookies are stored and sent with all future requests. It also randomizes the user agent. Future updates will automatically parallelize and throttle requests (*a great* *project to contribute to the data pipeline*).

Parsing JSON

In the event that your source of course data returns JSON, life is easy. You can find the fields and pull them out by simply treating the JSON as a python dictionary when the requester returns it.

Parsing HTML (or XML)

If, instead, your site is marked up with HTML, we use BeautifulSoup4 (BS4) to find certain divs and map the data inside of those divs to the fields of the ingestor.

Let's say the HTML looks like this:

```
<body>
    <div class="course-wrapper">
        <h1>EN.600.123</h1>
        <h4>Some Course Name</h4>
        <a href="urltosectiondata">More Info</a>
        ....
    </div>
    <div class="course-wrapper">
        ...
    </div>
    ...
</body>
```

We can then write the get courses function as follows:

```
def get_courses(self, department):
    soup = self.requester.get('urltothisdepartment.com')
    return soup.find_all(class_='course-wrapper')
```

And we can fill the ingestor based on these courses by:

```
courses = self.get_courses(department)
for course in courses:
    self.ingestor['course_code'] = course.find('h4').get_text()
    ...
```

To get section data, we can follow the "More Info" link and parse the resulting HTML in the same way:

```
section_html = self.requester.get(course.find('a')['href'])
```

Note: You can learn more about BS4 by reading their documentation . It is an extensive library that provides many excellent utilities for parsing HTML/XML.

Parse and Test

When you're ready you can go ahead and run your parser. You can do this by:

```
python manage.py ingest [SCHOOL_CODE]
```

Replacing SCHOOL_CODE with whatever your school's code (e.g. jhu) is. This will start the ingestion process, creating a file *data/courses.json* in your school's directory.

If, along the way, your ingestion fails to validate, the ingestor will throw useful errors to let you know how or why!

Once it runs to completion, you can *digest* the JSON, entering it into the database by running:

```
python manage.py digest [SCHOOL_CODE]
```

Note: To learn more, checkout the *Data Pipeline Documentation*

How to Run & Write Tests

Running Tests

Frontend

Run all tests:

```
npm test
```

Run single test:

```
npm test -- static/js/redux/__tests__/schema.test.js
```

Backend

Run all tests:

```
python manage.py test
```

Run all tests for a single app:

```
python manage.py test timetable
```

Run single test suite:

```
python manage.py test timetable.tests.UrlsTest
```

Run single test case:

```
python manage.py test timetable.tests.UrlTest.test_urls_call_correct_views
```

Run tests without resetting db:

```
python manage.py test -k
```

Our current test runner will only run db setup if the tests you're running touch the db.

Writing Tests

Unit Tests

Contributors are encouraged to write unit tests for changed or new code. By separating out logic into simple pure functions, you can isolate the behaviour you care about in your unit tests, and not worry about testing for side effects. Following the design principles outlined in the resources from the *Learning The Stack* section helps with this. For example, extracting all code that extract information from the state into selectors, which are pure functions that take the state (or some part of it) as input and output some data, will make it easy to test and change any state related behavior. Sometimes you may want to test behaviour that can't be extracted into a pure function, or that touches external interfaces. There are a number of strategies you can use in these cases.

Integration Tests

In the frontend, for testing the logic for rendering a component, look into snapshot tests. For testing async (thunk) action creators, our current tests create a store with desired initial state, dispatch the action, and then check that the action had the desired effect on the state. Backend requests are mocked using the nock library.

For testing views, we use django's built in client to send requests to the backend. It's also possible to use django's request factory to create requests to provide directly as input to your views.

End to End Tests

As the name implies, end to end tests test the entire app at once by simulating a semesterly user. When writing or changing end to end tests, it is recommended to familiarize yourself with the methods provided in SeleniumTestCase, which make it easy to perform certain actions on the app.

Backend Documentation

Timetable App

The timetable app is the core application that has been a part of Semester.ly since our very first release. The timetable app does the heavy lifting for timetable generation, sharing, and viewing.

Models

class `timetable.models.`**`Course`**`(`**args*, ***kwargs*`)`

Represents a course at a school, made unique by its course code. Courses persist across semesters and years. Their presence in a semester or year is indicated by the existence of sections assigned to that course for that semester or year. This is why a course does not have fields like professor, those varies.

The course model maintains only attributes which tend not to vary across semesters or years.

A course has many *`Section`* which a student can enroll in.

`school`

`CharField` – the school code corresponding to the school for the course

`code`

`CharField` – the course code without indication of section (E.g. EN.600.100)

`name`

`CharField` – the general name of the course (E.g. Calculus I)

description
`TextField` – the explanation of the content of the courzse

notes
`TextField`, optional – usually notes pertaining to registration (e.g. Lab Fees)

info
`TextField`, optional – similar to notes

unstopped_description
`TextField` – automatically generated description without stopwords

campus
`CharField`, optional – an indicator for which campus the course is taught on

prerequisites
`TextField`, optional – courses required before taking this course

corequisites
`TextField`, optional – courses required concurrently with this course

exclusions
`TextField`, optional – reasons why a student would not be able to take this

num_credits
`FloatField` – the number of credit hours this course is worth

areas
`CharField` – comma seperated list of all degree areas this course satisfies

department
`CharField` – department offering course (e.g. Computer Science)

level
`CharField` – indicator of level of course (e.g. 100, 200, Upper, Lower, Grad)

cores
`CharField` – core areas satisfied by this course

geneds
`CharField` – geneds satisfied by this course

related_courses
`ManyToManyField` of *Course*, optional – courses computed similar to this course

same_as
`ForeignKey` – If this course is the same as another course, provide Foreign key

vector
`PickleObjectField` – the vector representation of a course transformed from course vectorizer

get_avg_rating()
Calculates the avg rating for a course, -1 if no ratings. Includes all courses that are marked as the same by the self.same_as field on the model nstance.

Returns the average course rating

Return type (`float`)

get_reactions(*student=None*)
Return a list of dicts for each type of reaction (by title) for this course. Each dict has:

title: the title of the reaction

count: number of reactions with this title that this course has received

reacted: True if the student provided has given a reaction with this title

class `timetable.models.`**`CourseIntegration`**(*id*, *course*, *integration*, *json*)

class `timetable.models.`**`Evaluation`**(**args*, ***kwargs*)

A review of a course represented as a score out of 5, a summary/comment, along with the professor and year the review is in subject of.

course (`ForeignKey` to `Course`): the course this evaluation belongs to

score (`FloatField`): score out of 5.0 summary (`TextField`): text with information about why the rating was given professor (`CharField`): the professor(s) this review pertains to year (`CharField`): the year of the review course_code (`Charfield`): a string of the course code, along with section indicator

class `timetable.models.`**`Integration`**(*id*, *name*)

class `timetable.models.`**`Offering`**(**args*, ***kwargs*)

An Offering is the most granular part of the Course heirarchy. An offering may be looked at as the backend equivalent of a single slot on a timetable. For each day/time which a section meets, an offering is created.abs

`section`

`ForeignKey` to *`Section`* – the section which is the parent of this offering

`day`

`CharField` – the day the course is offered (single character M,T,W,R,F,S,U)

`time_start`

`CharField` – the time the slot starts in 24hrs time in the format (HH:MM) or (H:MM)

`time_end`

`CharField` – the time it ends in 24hrs time in the format (HH:MM) or (H:MM)

`location`

`CharField`, optional – the location the course takes place, defaulting to TBA if not provided

class `timetable.models.`**`Section`**(**args*, ***kwargs*)

Represents one (of possibly many) choice(s) for a student to enroll in a *`Course`* for a specific semester. Since this model is specific to a semester, it contains enrollment data, instructor information, textbooks, etc.

A section can come in different forms. For example, a lecture which is required for every student. However, it can also be a tutorial or practical. During timetable generation we allow a user to select one of each, and we can automatically choose the best combonation for a user as well.

A section has many offerings related to it. For example, section 1 of a *`Course`* could have 3 offerings (one that meets each day: Monday, Wednesday, Friday). Section 2 of a *`Course`* could have 3 other offerings (one that meets each: Tuesday, Thursday).

`course`

`Course` – The course this section belongs to

`meeting_section`

`CharField` – the name of the section (e.g. 001, L01, LAB2)

`size`

`IntegerField` – the capacity of the course (the enrollment cap)

`enrolment`

`IntegerField` – the number of students registered so far

`waitlist`

`IntegerField` – the number of students waitlisted so far

`waitlist_size`

`IntegerField` – the max size of the waitlist

section_type
CharField – the section type, example 'L' is lecture, 'T' is tutorial, *P* is practical

instructors
CharField – comma seperated list of instructors

semester
ForeignKey to *Semester* – the semester for the section

textbooks
ManyToManyField of *Textbook* – textbooks for this section via the *TextbookLink* model

was_full
BooleanField – whether the course was full during the last parse

get_textbooks()
Returns the textbook info using *tb.get_info()* for each textbook

class timetable.models.**Semester**(**args*, ***kwargs*)
Represents a semester which is composed of a name (e.g. Spring, Fall) and a year (e.g. 2017).

name
CharField – the name (e.g. Spring, Fall)

year
CharField – the year (e.g. 2017, 2018)

class timetable.models.**Textbook**(**args*, ***kwargs*)
A textbook which is associated with sections of courses. Stores information from the Amazon product API including a detail url and ISBN.

isbn
BigIntegerField – the primary (unique) key ISBN number

detail_url
URLField – url to the detail page on Amazon.com

image_url
URLField – url to product image hosted on Amazon.com

author
CharField – authors first and last name

title
CharField – the title of the book

class timetable.models.**TextbookLink**(**args*, ***kwargs*)
This model serves as a ManyToMany link betwen a *Section* anda textbook. The reason for this additional model is because the edge that connects a *Section* has a label which is whether that textbook is required. Thus, a seperate model/table exists to link the two with this label.abs

textbook
ForeignKey to *Textbook* – the textbook

is_required
BooleanField – whether or not the textbook is required

section
Section – the section the textbook is linked to

Views

class `timetable.views.`**`TimetableLinkView`**(***kwargs*)

A subclass of `FeatureFlowView` (see *Flows Documentation*) for the viewing of shared timetable links. Provides the logic for preloading the shared timetable into initData when a user hits the corresponding url. The frontend can then act on this data to load the shared timetable for viewing.

Additionally, on POST provides the functionality for the creation of shared timetables.

`get_feature_flow`(*request*, *slug*)

Overrides `FeatureFlowView` *get_feature_flow* method. Takes the slug, decrypts the hashed database id, and either retrieves the corresponding timetable or hits a 404.

`post`(*request*)

Creates a `SharedTimetable` and returns the hashed database id as the slug for the url which students then share and access.

class `timetable.views.`**`TimetableView`**(***kwargs*)

This view is responsible for responding to any requests dealing with the generation of timetables and the satisfaction of constraits provided by the frontend/user.

`post`(*request*)

Generate best timetables given the user's selected courses

Serializers

Utils

class `timetable.utils.`**`DisplayTimetable`**(*slots*, *has_conflict*, *name=''*, *events=None*, *id=None*)

Object that represents the frontend's interpretation of a timetable.

classmethod **`from_model`**(*timetable*)

Create DisplayTimetable from Timetable instance.

class `timetable.utils.`**`Slot`**(*course*, *section*, *offerings*, *is_optional*, *is_locked*)

`course`

Alias for field number 0

`is_locked`

Alias for field number 4

`is_optional`

Alias for field number 3

`offerings`

Alias for field number 2

`section`

Alias for field number 1

class `timetable.utils.`**`Timetable`**(*courses*, *sections*, *has_conflict*)

`courses`

Alias for field number 0

`has_conflict`

Alias for field number 2

sections

Alias for field number 1

timetable.utils.**add_meeting_and_check_conflict**(*day_to_usage*, *new_meeting*, *school*)

Takes a @day_to_usage dictionary and a @new_meeting section and returns a tuple of the updated day_to_usage dict and a boolean which is True if conflict, False otherwise.

timetable.utils.**courses_to_slots**(*courses*, *locked_sections*, *semester*, *optional_course_ids*)

Return a list of lists of Slots. Each Slot sublist represents the list of possibilities for a given course and section type, i.e. a valid timetable consists of any one slot from each sublist.

timetable.utils.**find_slots_to_fill**(*start*, *end*, *school*)

Take a @start and @end time in the format found in the coursefinder (e.g. 9:00, 16:30), and return the indices of the slots in thet array which represents times from 8:00am to 10pm that would be filled by the given @start and @end. For example, for uoft input: '10:30', '13:00' output: [5, 6, 7, 8, 9]

timetable.utils.**get_current_semesters**(*school*)

List of semesters ordered by academic temporality.

For a given school, get the possible semesters ordered by the most recent year for each semester that has course data, and return a list of (semester name, year) pairs.

timetable.utils.**get_day_to_usage**(*custom_events*, *school*)

Initialize day_to_usage dictionary, which has custom events blocked out.

timetable.utils.**get_hour_from_string_time**(*time_string*)

Get hour as an int from time as a string.

timetable.utils.**get_hours_minutes**(*time_string*)

Return tuple of two integers representing the hour and the time given a string representation of time. e.g. '14:20' -> (14, 20)

timetable.utils.**get_minute_from_string_time**(*time_string*)

Get minute as an int from time as a string.

timetable.utils.**get_time_index**(*hours*, *minutes*, *school*)

Take number of hours and minutes, and return the corresponding time slot index

timetable.utils.**get_xproduct_indicies**(*lists*)

Takes a list of lists and returns two lists of indicies needed to iterate through the cross product of the input.

timetable.utils.**slots_to_timetables**(*slots*, *school*, *custom_events*, *with_conflicts*)

Generate timetables in a depth-first manner based on a list of slots.

timetable.utils.**update_locked_sections**(*locked_sections*, *cid*, *locked_section*, *semester*)

Take cid of new course, and locked section for that course and toggle its locked status (ie if was locked, unlock and vice versa.

Courses App

The courses app deals with the accesing course information, the sharing of courses, and the rendering of the course/all course pages.

Views

class courses.views.**CourseDetail**(***kwargs*)

View that handles individual course entities.

get (*request*, *sem_name*, *year*, *course_id*)
Return detailed data about a single course. Currently used for course modals.

class `courses.views.`**CourseModal** (***kwargs*)
A `FeatureFlowView` for loading a course share link which directly opens the course modal on the frontend. Therefore, this view overrides the *get_feature_flow* method to fill intData with the detailed course json for the modal.abs

Saves a `SharedCourseView` for analytics purposes.

`courses.views.`**all_courses** (*request*, **args*, ***kwargs*)
Generates the full course directory page. Includes links to all courses and is sorted by department.

`courses.views.`**course_page** (*request*, **args*, ***kwargs*)
Generates a static course page for the provided course code and school (via subdomain). Completely outside of the React framework purely via Django templates.

`courses.views.`**get_classmates_in_course** (*request*, *school*, *sem_name*, *year*, *course_id*)
Finds all classmates for the authenticated user who also have a timetable with the given course.

Utils

`courses.utils.`**get_sections_by_section_type** (*course*, *semester*)
Return a map from section type to Sections for a given course and semester.

`courses.utils.`**sections_are_filled** (*sections*)
Return True if all sections are filled beyond their max enrollment.

Serializers

class `courses.serializers.`**CourseSerializer** (*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Serialize a Course into a dictionary with detailed information about the course, and all related entities (eg Sections). Used for search results and course modals. Takes a context with parameters: school: str (required) semester: Semester (required) student: Student (optional)

get_evals (*course*)
Flag all eval instances s.t. there exists repeated term+year values. :returns: List of modified evaluation dictionaries (added flag 'unique_term_year')

get_popularity_percent (*course*)
Return percentage of course capacity that is filled by registered students.

get_regexed_courses (*course*)
Given course data, search for all occurrences of a course code in the course description and prereq info and return a map from course code to course name for each course code.

`courses.serializers.`**get_section_dict** (*section*)
Returns a dictionary of a section including indicator of whether that section is filled

Student App

The Student model is an abstraction over the Django user to provide us with a more full user profile including information pulled from social authentication via Google and/or Facebook. This app handles utilities for overriding the Python Social Auth authentication pipeline, while also handling the functionality for logged in users.

The student app also encapsulates all models tied directly to a user like PersonalTimetables, PersonalEvents, Reactions, and notification tokens.

Models

Models pertaining to Students.

class `student.models.`**`PersonalEvent`**(**args*, ***kwargs*)

A custom event that has been saved to a user's PersonalTimetable so that it persists across refresh, device, and session. Marks when a user is not free. Courses are scheduled around it.abs

class `student.models.`**`PersonalTimetable`**(**args*, ***kwargs*)

Database object representing a timetable created (and saved) by a user.

A PersonalTimetable belongs to a Student, and contains a list of Courses and Sections that it represents.

class `student.models.`**`Reaction`**(**args*, ***kwargs*)

Database object representing a reaction to a course.

A Reaction is performed by a Student on a Course, and can be one of REACTION_CHOICES below. The reaction itself is represented by its *title* field.

class `student.models.`**`RegistrationToken`**(**args*, ***kwargs*)

A push notification token for Chrome noitification via Google Cloud Messaging

class `student.models.`**`Student`**(**args*, ***kwargs*)

Database object representing a student.

A student is the core user of the app. Thus, a student will have a class year, major, friends, etc. An object is only created for the user if they have signed up (that is, signed out users are not represented by Student objects).

Views

class `student.views.`**`ClassmateView`**(***kwargs*)

Handles the computation of classmates for a given course, timetable, or simply the count of all classmates for a given timetable.

`get`(*request*, *sem_name*, *year*)

Returns

If the query parameter 'count' is present Information regarding the number of friends only:

```
{
    "id": Course with the most friends,
    "count": The maximum # of friends in a course,
    "total_count": the total # in all classes on timetable,
}
```

If the query parameter course_ids is present a list of dictionaries representing past classmates and current classmates. These are students who the authenticated user is friends with and who has social courses enabled.:

```
[{
    "course_id":6137,
    "past_classmates":[...],
    "classmates":[...]
}, ...]
```

Otherwise a list of friends and non-friends alike who have social_all enabled to be dispalyed in the "find-friends" modal. Sorted by the number courses the authenticated user shares.:

```
[{
    "name": "...",
    "is_friend": Whether or not the user is current user's friend,
    "profile_url": link to FB profile,
    "shared_courses": [...],
    "peer": Info about the user,
}, ...]
```

class `student.views.`**`GCalView`**(***kwargs*)

Handles interactions with the Google Calendar API V3 for pulling and/or sending calendars and calendar events.

`post`(*request*)

Takes the timetable in request.body and creates a weekly recurring event on Google calendar for each slot in a given week. Names the Google Calendar "Semester.ly Schedule" if unnamed, otherwise "[Timetable Name] - Semester.ly".

class `student.views.`**`ReactionView`**(***kwargs*)

Manages the creation of Reactions to courses.

`post`(*request*)

Create a Reaction for the given course id, with the given title matching one of the possible emojis. If already present, remove that reaction.

class `student.views.`**`UserTimetableView`**(***kwargs*)

Responsible for the viewing and managing of all Students' `PersonalTimetable`.

`delete`(*request*, *sem_name*, *year*, *tt_name*)

Deletes a PersonalTimetable by name/year/term.

`get`(*request*, *sem_name*, *year*)

Returns student's personal timetables

`post`(*request*)

Duplicates a personal timetable if a 'source' is provided. Else, creates a personal timetable based on the courses, custom events, preferences, etc. which are provided.

`update_events`(*tt*, *events*)

Replace tt's events with input events. Deletes all old events to avoid buildup in db

class `student.views.`**`UserView`**(***kwargs*)

Handles the accessing and mutating of user information and preferences.

`delete`(*request*)

Delete this user and all of its data

`get`(*request*)

Renders the user profile/stats page which indicates all of a student's reviews of courses, what social they have connected, whether notificaitons are enabled, etc.

`patch`(*request*)

Updates a user settings to match the corresponding values passed in the request body. (e.g. social_courses, class_year, major)

`student.views.`**`accept_tos`**(*request*)

Accepts the terms of services for a user, saving the `datetime` the terms were accepted.

`student.views.`**`create_unsubscribe_link`**(*student*)

Generates a unsubscribe link which directs to the student unsubscribe view.

`student.views.`**`get_friend_count_from_course_id`**(*school*, *student*, *course_id*, *semester*)

Computes the number of friends a user has in a given course for a given semester.

Ignores whether or not those friends have social courses enabled. Never exposes those user's names or infromation. This count is used purely to upsell user's to enable social courses.

`student.views.`**`log_ical_export`**(**args*, ***kwargs*)

Logs that a calendar was exported on the frotnend and indicates it was downloaded rather than exported to Google calendar.

`student.views.`**`unsubscribe`**(*request*, *student_id*, *token*)

If the student matches the token and the tokens is valid , unsubscribes user from emails marking student.emails_enabled to false. Redirects to index.

Utils

`student.utils.`**`get_classmates_from_course_id`**(*school*, *student*, *course_id*, *semester*, *friends=None*, *include_same_as=False*)

Get's current and past classmates (students with timetables containing the provided course ID). Classmates must have social_courses enabled to be included. If social_sections is enabled, info about what section they are in is also passed.

Parameters

- **`school`** (`str`) – the school code (e.g. 'jhu')
- **`student`** (`Student`) – the student for whom to find classmates
- **`course_id`** (`int`) – the database id for the course
- **`semester`** (`Semester`) – the semester that is current (to check for)
- **`friends`** (`list` of `Students`) – if provided, does not re-query for friends list, uses provided list.
- **`include_same_as`** (`bool`) – If provided as true, searches for classmates in any courses marked as "same as" in the database.

`student.utils.`**`get_classmates_from_tts`**(*student*, *course_id*, *tts*)

Returns a list of classmates a student has from a list of other user's timetables. This utility does the leg work for *`get_classmates_from_course_id()`* by taking either a list of current or past timetables and finding classmates relevant to that list.

If both students have social_offerings enabled, adds information about what sections the student is enrolled in on each classmate.

`student.utils.`**`get_student`**(*request*)

Returns the student belonging to the authenticated user

Return type (`Student`)

`student.utils.`**`get_student_tts`**(*student*, *school*, *semester*)

Returns serialized list of a student's `PersonalTimetable` objects ordered by last updated for passing to the frontend.

`student.utils.`**`next_weekday`**(*d*, *weekday*)

Given a current date, d, and a target weekday, calculate the next occurence (moving in the future) of that weekday.

Returns the next weekday of the given type

Return type (`datetime.datetime`)

Serializers

`student.serializers.`**`get_student_dict`**`(`*school*, *student*, *semester*`)`

Return serialized representation of a student.

Searches App

Searches app provides a useful, efficient and scalable search backend using basic techinques of information retrieval.

Each course model contains a vector stored as a pickled scipy sparse vector. This vector represents this course. Upon search, a vectorizer creates a similar vector representation for that query. We then fetch ~100 candidates for serving as search results. These canidates are then sorted by the cosine similarity between their vector and the query vector.

To increase accuracy and provide for a clean search experience for key use cases, the search places a heavier weight on courses with matching titles. However, description and other fields are searched as well.

Views

class `searches.views.`**`CourseSearchList`**`(`***kwargs*`)`

Course Search List.

`get``(`*request*, *query*, *sem_name*, *year*`)`

Return vectorized search results.

`post``(`*request*, *query*, *sem_name*, *year*`)`

Return advanced search results.

Utils

class `searches.utils.`**`Searcher`**

Searcher class implements baseline search and vectorized search based on information retrieval techniques.

`get_acronym``(`*name*`)`

Returns an acronym of a course name.

`get_cosine_sim``(`*sparse_vec1*, *sparse_vec2*`)`

Computes cosine similarity between two sparse vectors.

`get_most_relevant_filtered_courses``(`*query*, *course_filtered*`)`

Returns the most relevant filtered courses given a query from filtered course objects.

`get_score``(`*course*, *query*, *query_vector*`)`

Computes similarity score based on cosine similarity and match between query and course name.

`get_similarity``(`*query*, *course*`)`

Vectorizes query and returns a cosine similarity score between query and course vector.

`load_count_vectorizer``()`

Loads english dictionary count vectorizer pickle object.

`matches_name``(`*query*, *course_name*`)`

Returns a score (2, 1, 0) of a query match to course name.

print_similiarity_scores(*courses*, *query*)
Prints all course similarity scores given a query (for debugging).

vectorize_query(*query*)
Vectorizes a user's query using count vectorizer.

vectorized_search(*school*, *query*, *semester*)
Returns filtered courses that are most relevant to a given query.

wordify(*course_vector*)
Converts a course vector back into string using count vectorizer.

class searches.utils.**Vectorizer**
Vectorizer class creates a dictionary over courses and build course vectors using count vectorizer.

course_to_str(*name*, *description*, *area*, *weight*)
Returns a string representation of a course using a Porter Stemmer.

doc_to_lower_stem_str(*doc*)
Converts words in document(string) to lowercase, stemmed words.

vectorize()
Vectorize function transforms and saves entire course objects into course vectors using TF-IDF.

searches.utils.**baseline_search**(*school*, *query*, *semester*)
Baseline search returns courses that are contained in the name from a query (legacy code).

searches.utils.**course_desc_contains_token**(*token*)
Returns a query set of courses where tokens are contained in descriptions.

searches.utils.**course_name_contains_token**(*token*)
Returns a query set of courses where tokens are contained in code or name.

Exams App

The exams app provides minimal infrastructure for concluding exam periods based on a timetable. So far it only supports rule based scheduling meaning "If Monday 9-12, exam is 5/12 2-5pm" and is only used at Johns Hopkins. However, the infrastructure can be used for any school using a rule based approach.

Models

class exams.models.**FinalExamShare**(**args*, ***kwargs*)
Database object representing a shared final exam schedule. A final exam schedule belongs to a Student and contains the list of classes which the user needs to check finals for

Views

Final Exam Scheduler

Example Implementation

class exams.jhu_final_exam_scheduler.**JHUFinalExamScheduler**
Database Object that has a list of JHU's rules for the current semester. Should be updated every semester. Initialize each Rule with select fields depending on what determines whether the Rule is valid for. See Final Exam Scheduler for more information

Agreement App

In order to use our system, users must agree to our privacy policy/terms and conditions.

When a user is not logged in, this is done implicitly (no click to accept is required). Out of respect for our users and to be fully transparent, we surface a banner when the user is not logged in to bring this implicity agreement to their attention.

When a user is logged in, the agreement must be explicity. During signup the user is prompted to agree to the terms and must do so in order to continue using the application. If the documents have been updated since the user last agreed, they will be notified of this change and once again asked to agree to the updated terms/policy.

Models

class `agreement.models.`**`Agreement`**(**args*, ***kwargs*)
    Database object representing updates to the ToS/privacy policy.

E2E Test Utils

class `semesterly.test_utils.`**`SeleniumTestCase`**(**args*, ***kwargs*)
    This test case extends the Django StaticLiveServerTestCase. It creates a selenium ChromeDriver instance on setUp of each test. It navigates to the live url for the static live server. It also provides utilities and assertions for navigating and testing presence of elements or behavior.

`img_dir`
    str – Directory to save screenshots on failure.

`driver`
    WebDriver – Chrome WebDriver instance.

`timeout`
    int – Socket default timeout.

`add_course`(*course_idx*, *n_slots*, *n_master_slots*, *by_section=''*, *code=None*)
    Adds a course via search results and asserts the corresponding number of slots are found

Parameters

- **`course_idx`** (*`int`*) – index into the search results corresponding the to course to add
- **`n_slots`** (*`int`*) – the number of slots expected after add
- **`n_master_slots`** (*`int`*) – the number of master slots expected after add
- **`by_section`** (*`str, optional`*) – if provided adds the specific section of the course
- **`code`** (*`str, optional`*) – the course code to add, validates presence if provided

`add_course_from_course_modal`(*n_slots*, *n_master_slots*)
    Adds a course via the course modal action. Requires that the course modal be open.

`allow_conflicts_add`(*n_slots*)
    Allows conflicts via the conflict alert action, then validates that the course was added

`assert_friend_image_found`(*friend*)
    Asserts that the provided friend's image is found on the page

`assert_friend_in_modal`(*friend*)
    Asserts that the provided friend's image is found on the modal

assert_invisibility(*locator*, *root=None*)
Asserts the invisibility of the provided element

Parameters

- **locator** – A tuple of (By.*, 'indentifier')
- **root** (`bool, optional`) – The root element to search from, root of DOM if None

assert_loader_completes()
Asserts that the semester.ly page loader has completed

assert_n_elements_found(*locator*, *n_elements*, *root=None*)
Asserts that n_elements are found by the provided locator

assert_ptt_const_across_refresh()
Refreshes the browser and asserts that the tuple version of the personal timetable is equivalent to pre-refresh

assert_ptt_equals(*ptt*)
Asserts equivalency between the provided ptt tuple and the current ptt

assert_slot_presence(*n_slots*, *n_master_slots*)
Assert n_slots and n_master_slots are on the page

change_ptt_name(*name*)
Changes personal timetable name to the provided title

change_term(*term*, *clear_alert=False*)
Changes the term to the provided term by matching the string to the string found in the semester dropdown on Semester.ly

clear_tutorial()
Clears the tutorial modal for first time users

click_off()
Clears the focus of the driver

close_course_modal()
Closes the course modal using the (x) button

complete_user_settings_basics(*major*, *class_year*)
Completes major/class year/TOS agreement via the welcome modal

Parameters

- **major** (`str`) – Student's major
- **class_year** (`str`) – Student's class year

create_friend(*first_name*, *last_name*, ***kwargs*)
Creates a friend of the primary (first) user

create_personal_timetable_obj(*friend*, *courses*, *semester*)
Creates a personal timetable object belonging to the provided user with the given courses and semester

create_ptt(*name=None*)
Create a personaltimetable with the provided name when provided

description(**args*, ***kwds*)
A context manager which wraps a group of code and adds details to any exceptions thrown by the enclosed lines. Upon such an exception, the context manager will also take a screenshot of the current state of self.driver, writing a PNG to self.img_dir, labeled by the provided description and a timetstamp.

enter_search_query(*query*)
Enters the provided query into the search box

execute_action_expect_alert(*action*, *alert_text_contains=''*)
Executes the provided action, asserts that an alert appears and validates that the alert text contains the provided string (when provided)

find(*locator*, *get_all=False*, *root=None*, *clickable=False*, *hidden=False*)
Locates element in the DOM and returns it when found.

Parameters

- **locator** – A tuple of (By.*, 'indentifier')
- **get_all** (*bool, optional*) – If true, will return list of matching elements
- **root** (*bool, optional*) – The root element to search from, root of DOM if None
- **clickable** (*bool, optional*) – If true, waits for clickability of element
- **hidden** (*bool, optional*) – If true, will allow for hidden elements

Returns The WebElement object returned by self.driver (Selenium)

follow_and_validate_url(*url*, *validate*)
Opens a new window, switches to it, gets the url and validates it using the provided validating function.

Parameters

- **url** (*str*) – the url to follow and validate
- **validate** (*func*) – the function which validates the new page

follow_share_link_from_slot()
Click the share link on the slot and follow it then validate the course modal

get_elements_as_text(*locator*)
Gets elements using self.get and represents them as text

get_test_url(*school*, *path=''*)
Get's the live server testing url for a given school.

Parameters

- **school** (*str*) – the string for which to create the test url
- **path** (*str*) – the appended path to file or page with trailing /

Returns the testing url

init_screenshot_dir()
Initializes directory to which we store test failure screenshots

lock_course()
Locks the first course on the timetable

login_via_fb(*email*, *password*)
Login user via fb by clicking continue with Facebook in the signup modal, entering the user's credentials into Facebook, then returns to Semester.ly

Parameters

- **email** (*str*) – User's email
- **password** (*str*) – User's password

login_via_google(*email*, *password*, ***kwargs*)

Mocks the login of a user via Google by clicking continue with Facebook in the signup modal. Then manually creates and logins a user. All kwargs are passed to the user model on creation (e.g. name and email).

Parameters

- **email** (*str*) – User's email
- **password** (*str*) – User's password

open_and_query_adv_search(*query*, *n_results=None*)

Open's the advanced search modal and types in the provided query, asserting that n_results are then returned

open_course_modal_from_search(*course_idx*)

Opens course modal from search by search result index

open_course_modal_from_slot(*course_idx*)

Opens the course modal from the nth slot

ptt_to_tuple()

Converts personal timetable to a tuple representation

remove_course(*course_idx*, *from_slot=False*, *n_slots_expected=None*)

Removes a course from the user's timetable, asserts master slot is removed.

Parameters

- **course_idx** (*int*) – the index of the course for which to remove
- **from_slot** (*bool, optional*) – if provided, removes via slot rather than via a master_slot
- **n_slots_expected** (*int, optional*) – if provided, asserts n slots found after removal

remove_course_from_course_modal(*n_slots_expected=None*)

Removes course via the action within the course's course modal. Requires that the course modal be open.

save_ptt()

Saves the user's current personal timetable and returs a tuple representation

save_user_settings()

Saves user setttings by clicking the button, asserts that the modal is then invisible

search_course(*query*, *n_results*)

Searches a course and asserts n_results elements are found

select_nth_adv_search_result(*index*, *semester*)

Selects the nth advanced search result with a click. Validates the course modal body displayed in the search reuslts

share_timetable(*courses*)

Clicks the share button via the top bar and validates it. Validation is done by following the url and checking the timetable using the validate_timetable function

switch_to_ptt(*name*)

Switches to the personal timetable with matching name

take_alert_action()

Takes the action provided by the alert by clicking the button on when visible

validate_course_modal()
Validates the course modal displays proper course data

validate_course_modal_body(*course*, *modal*, *semester*)
Validates the course modal body displays credits, name, code, etc.

validate_timeable(*courses*)
Validate timetable by checking that for each course provided, a slot exists with that course's name and course code.

semesterly.test_utils.**force_login**(*user*, *driver*, *base_url*)
Forces the login of the provided user setting all cookies. Function will refresh the provided drivfer and the user will be logged in to that session.

class semesterly.test_utils.**function_returns_true**(*func*)
An expectation for checking if the provided function returns true

class semesterly.test_utils.**n_elements_to_be_found**(*locator*, *n_*)
An expectation for checking if the n elements are found locator, text

class semesterly.test_utils.**text_to_be_present_in_element_attribute**(*locator*, *text_*, *attribute_*)
An expectation for checking if the given text is present in the element's locator, text

class semesterly.test_utils.**text_to_be_present_in_nth_element**(*locator*, *text_*, *index_*)
An expectation for checking if the given text is present in the nth element's locator, text

class semesterly.test_utils.**url_matches_regex**(*pattern*)
Expected Condition which waits until the browser's url matches the provided regex

Helpers App

Decorators

helpers.decorators.**validate_subdomain**(*view_func*)
Validates subdomain, redirecting user to index iof the school is invalid.

Mixins

class helpers.mixins.**FeatureFlowView**(***kwargs*)
Template that handles GET requests by rendering the homepage. Feature_name or get_feature_flow() can be overridden to launch a feature or action on homepage load.

get_feature_flow(*request*, **args*, ***kwargs*)
Return data needed for the feature flow for this HomeView. A name value is automatically added in .get() using the feature_name class variable. A semester value can also be provided, which will change the initial semester state of the home page.

class helpers.mixins.**ValidateSubdomainMixin**
Mixin which validates subdomain, redirecting user to index if the school is not in `ACTIVE_SCHOOLS`.

Authentication Pipeline

Views

class authpipe.views.**RegistrationTokenView**(***kwargs*)

Handles registration and deletion of tokens for maintaining chrome notifications for users who choose to enable the feature.

put(*request*)

Creates a notification token for the user.

Utils

authpipe.utils.**associate_students**(*strategy*, *details*, *response*, *user*, **args*, ***kwargs*)

Part of our custom Python Social Auth authentication pipeline. If a user already has an account associated with an email, associates that user with the new backend.

authpipe.utils.**check_student_token**(*student*, *token*)

Validates a token: checks that it is at most 2 days old and that it matches the currently authenticated student.

authpipe.utils.**create_student**(*strategy*, *details*, *response*, *user*, **args*, ***kwargs*)

Part of the Python Social Auth pipeline which creates a student upon signup. If student already exists, updates information from Facebook or Google (depending on the backend).

Saves friends and other information to fill database.

Flows Documentation

Initalization

When a user loads the home timetable page, `FeatureFlowView` inside of `timetable.utils` is used to handle the request. On initial page load, the frontend requires some data to initialize the redux state, like information about the current user, the list of possible semesters for the school, and the list of student integrations. This initial data is created inside of the view, and passed in as a single json string in the response context:

```
class FeatureFlowView(ValidateSubdomainMixin, APIView):

    def get(self, request, *args, **kwargs):
        # ...gather values for init_data

        init_data = {
            'school': self.school,
            'currentUser': get_user_dict(self.school, self.student, sem),
            'currentSemester': curr_sem_index,
            'allSemesters': all_semesters,
            'uses12HrTime': self.school in AM_PM_SCHOOLS,
            'studentIntegrations': integrations,
            'examSupportedSemesters': map(all_semesters.index,
                                          final_exams_available.get(self.
↪school, [])),

            'featureFlow': dict(feature_flow, name=self.feature_name)
        }

        return render(request, 'timetable.html', {'init_data': json.
↪dumps(init_data)})
```

which makes the `init_data` variable accessible in *timetable.html*. This dumped json string is then passed to the frontend as a global variable:

```
<script type="text/javascript">
  var initData = "{{init_data|escapejs}}";
</script>
```

And then parsed inside of the `setup()` function in `init.jsx`

```
const setup = () => (dispatch) => {
    initData = JSON.parse(initData);

    // pass init data into the redux state
    dispatch({ type: ActionTypes.INIT_STATE, data: initData });

    // do other logic with initData...
};
```

In other words, the data that the frontend requires is retrieved/calculated inside of `FeatureFlowView`, and then passed to the frontend as global variable `initData`. The frontend then does any logic it needs based on that data inside of `setup()` in `init.jsx`. Any data that needs to be reused later on from `initData` should be passed in to the redux state so that the only global variable uses appear in `setup()`.

Feature Flows

One such piece of data that is passed to the frontend is a `featureFlow` object. This object is obtained as the return value of `.get_feature_flow()`, in addition to a `name: self.feature_name` key value pair. In the default implementation, this is just the dictionary `{name: None}`:

```
class FeatureFlowView(ValidateSubdomainMixin, APIView):
    feature_name = None

    def get_feature_flow(self, request, *args, **kwargs):
        return {}

    def get(self, request, *args, **kwargs):
        ...
        feature_flow = self.get_feature_flow(request, *args, **kwargs)
        init_data = {
            ...
            'featureFlow': dict(feature_flow, name=self.feature_name)
        }

        return render(request, 'timetable.html', {'init_data': json.
↪dumps(init_data)})
```

This feature flow value can be used to store any extra information that the frontend needs for any endpoints that would require initial data to be loaded. For example, when loading a timetable share link, the frontend also needs to get data about the timetable that is being shared - instead of making a request to the backend after page load, this information can be provided by the backend directly by passing this information in the feature flow. It is easy to write new views that pass different data and have custom logic by subclassing `FeatureFlowView` and overwriting the `get_feature_flow()` method and the `.feature_name` class attribute.

Having this data all stored under the key `featureFlow` in `init_data` ensures two things. Firstly, it makes explicit that there can only be one feature flow in play at a time (we can't load a timetable share link and a course share link at the same time), and secondly, it allows the frontend to know where to look for any feature data and act accordingly. In practice, this is done by switching on the name of the feature flow:

```
const setup = () => (dispatch) => {
    initData = JSON.parse(initData);

    dispatch({ type: ActionTypes.INIT_STATE, data: initData });

    // do other logic with initData...

    dispatch(handleFlows(initData.featureFlow));
};

const handleFlows = featureFlow => (dispatch) => {
    switch (featureFlow.name) {
        case 'SIGNUP':
            dispatch({ type: ActionTypes.TRIGGER_SIGNUP_MODAL });
            break;
        case 'USER_ACQ':
            dispatch({ type: ActionTypes.TRIGGER_ACQUISITION_MODAL });
            break;
        case 'SHARE_TIMETABLE':
            dispatch({ type: ActionTypes.CACHED_TT_LOADED });
            dispatch(lockTimetable(featureFlow.sharedTimetable, true,␣
↪initData.currentUser.isLoggedIn));
            break;
        // ... etc.
        default:
            // unexpected feature name
            break;
  }
};
```

Example

To help understand how feature flows work, let's go through the code for an example feature flow: course sharing. In order to implement course sharing, we want to create a new view/endpoint that retrieves course data based on the url and passes it to the frontend, which would then update the redux state and dispatch an action to open the course modal.

We start be defining a new endpoint for this feature flow:

```
url(r'course/(?P<code>.+?)/(?P<sem_name>.+?)/(?P<year>.+?)/*$',
                    courses.views.CourseModal.as_view())
```

Then we create a new `FeatureFlowView` for this endpoint which needs to do two things: define a name for the feature flow, which the frontend look at to determine what action to do, and return the course data that the frontend needs inside of `get_feature_flow()`:

```
class CourseModal(FeatureFlowView):
    feature_name = "SHARE_COURSE"

    def get_feature_flow(self, request, code, sem_name, year):
        semester, _ = Semester.objects.get_or_create(name=sem_name,␣
↪year=year)
        code = code.upper()
        course = get_object_or_404(Course, school=self.school, code=code)
        course_json = get_detailed_course_json(self.school, course, semester,
↪ self.student)
```

```
        # analytics
        SharedCourseView.objects.create(
            student=self.student,
            shared_course=course,
        ).save()

        return {'sharedCourse': course_json, 'semester': semester}
```

The frontend can now add a new case in `handleFlows` to perform logic for this feature flow:

```
const handleFlows = featureFlow => (dispatch) => {
    switch (featureFlow.name) {
        ...
        case 'SHARE_COURSE':
            dispatch(setCourseInfo(featureFlow.sharedCourse));
            dispatch(fetchCourseClassmates(featureFlow.sharedCourse.id));
            break;
        // ... etc.
        default:
            // unexpected feature name
            break;
  }
};
```

Shortcuts

Some feature flows don't require any extra data - they simply require the frontend to know that a feature flow is being run. For example, for the signup feature flow, loading the page at `/signup` should simply open the signup modal, which requires no extra logic or data other than knowing that it should occur. We could do this by writing a new view:

```
class SignupModal(FeatureFlowView):
    feature_name = "SIGNUP"
```

We do not need to implement `.get_feature_flow()` since the frontend doesn't require any extra data and the default implementation already returns an empty dictionary. We can simplify this by simply declaring this view directly inside of the urls file:

```
url(r'^signup/*$/', FeatureFlowView.as_view(feature_name='SIGNUP')
```

see https://github.com/noahpresler/semesterly/pull/838 for the original pull request implementing feature flows

Data Pipeline Documentation

Semester.ly's data pipeline provides the infrastructure by which the database is filled with course information. Whether a given University offers an API or an online course catalogue, this pipeline lends developers an easy framework to work within to pull that information and save it in our Django Model format.

General System Workflow

1. Pull HTML/JSON markup from a catalogue/API
2. Map the fields of the mark up to the fields of our ingestor (by simply filling a python dictionary).
3. The ingestor preprocesses the data, validates it, and writes it to JSON.

4. Load the JSON into the database.

Note: This process happens automatically via Django/Celery Beat Periodict Tasks. You can learn more about these schedule tasks below (*Scheduled Tasks*).

Steps 1 and 2 are what we call **parsing** – an operation that is non-generalizable across all Universities. Often a new parser must be written. For more information on this, read *Add a School*.

Parsing Library Documentation

Base Parser

class `parsing.library.base_parser.`**`BaseParser`**(*school*, *config=None*, *output_path=None*, *output_error_path=None*, *break_on_error=True*, *break_on_warning=False*, *skip_duplicates=True*, *display_progress_bar=False*, *validate=True*, *tracker=None*)

Bases: `object`

Abstract base parser for data pipeline parsers.

`extractor`
`parsing.library.extractor.Extractor`

`ingestor`
`parsing.library.ingestor.Ingestor`

`requester`
`parsing.library.requester.Requester`

`school`
`str` – School that parser is for.

`end`()
Finish the parse.

`start`(***kwargs*)
Start the parse.

Parameters **`**kwargs`** – expanded in child parser.

Requester

class `parsing.library.requester.`**`Requester`**

Bases: `object`

`get`(*url*, *params=’‘*, *session=None*, *cookies=None*, *headers=None*, *verify=True*, ***kwargs*)
HTTP GET.

Parameters

- **`url`** (*`str`*) – url to query
- **`params`** (*`dict`*) – payload dictionary of HTTP params (default None)
- **`cookies`** (*`None, optional`*) – Description

- **headers** (*None, optional*) – Description
- **verify** (*bool, optional*) – Description
- ****kwargs** – Description

Examples

TODO

http_request (*do_http_request, type, parse=True, quiet=True, timeout=60, throttle=<function <lambda>>*)

Perform HTTP request.

Parameters

- **do_http_request** – function that returns request object
- **type** (*str*) – GET, POST, HEAD
- **parse** (*bool, optional*) – Specifies if return should be parsed. Autodetects parse type as html, xml, or json.
- **quiet** (*bool, optional*) – suppress output if True (default True)
- **timeout** (*int, optional*) – Description
- **throttle** (*lambda, optional*) – Description

Returns if parse is False soup: soupified/jsonified text of http request

Return type request object

static **markup** (*response*)

Autodects html, json, or xml format in response.

Parameters **response** – raw response object

Returns markedup response

new_user_agent ()

overwrite_header (*new_headers*)

post (*url, data='', params='', cookies=None, headers=None, verify=True, **kwargs*)

HTTP POST.

Parameters

- **url** (*str*) – url to query
- **data** (*str, optional*) – HTTP form key-value dictionary
- **params** (*dict*) – payload dictionary of HTTP params
- **cookies** (*None, optional*) – Description
- **headers** (*None, optional*) – Description
- **verify** (*bool, optional*) – Description
- ****kwargs** – Description

Ingestor

exception `parsing.library.ingestor.`**`IngestionError`**(*data*, **args*)

Bases: *`parsing.library.exceptions.PipelineError`*

Ingestor error class.

`args`

`message`

exception `parsing.library.ingestor.`**`IngestionWarning`**(*data*, **args*)

Bases: *`parsing.library.exceptions.PipelineWarning`*

Ingestor warning class.

`args`

`message`

class `parsing.library.ingestor.`**`Ingestor`**(*config*, *output*, *break_on_error=True*, *break_on_warning=False*, *display_progress_bar=True*, *skip_duplicates=True*, *validate=True*, *tracker=<parsing.library.tracker.NullTracker object>*)

Bases: `dict`

Ingest parsing data into formatted json.

Mimics functionality of dict.

`ALL_KEYS`
set – Set of keys supported by Ingestor.

`break_on_error`
bool – Break/cont on errors.

`break_on_warning`
bool – Break/cont on warnings.

`school`
str – School code (e.g. jhu, gw, umich).

`skip_duplicates`
bool – Skip ingestion for repeated definitions.

`tracker`
library.tracker – Tracker object.

`UNICODE_WHITESPACE`
TYPE – regex that matches Unicode whitespace.

`validate`
bool – Enable/disable validation.

`validator`
library.validator – Validator instance.

`ALL_KEYS` = set(['school_subdivision_code', 'code', 'isbn', 'author', 'prerequisites', 'instr', 'meetings', 'year', 'time_end'

`clear`() → None. Remove all items from D.

`copy`() → a shallow copy of D

end()
Finish ingesting.

Close i/o, clear internal state, write meta info

fromkeys(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

ingest_course()
Create course json from info in model map.

Returns course

Return type dict

ingest_eval()
Create evaluation json object.

Returns eval

Return type dict

ingest_meeting(*section*, *clean_only=False*)
Create meeting ingested json map.

Parameters **section** (*dict*) – validated section object

Returns meeting

Return type dict

ingest_section(*course*)
Create section json object from info in model map.

Parameters **course** (*dict*) – validated course object

Returns section

Return type dict

ingest_textbook()
Create textbook json object.

Returns textbook

Return type dict

ingest_textbook_link(*section=None*)
Create textbook link json object.

Parameters **section** (None, dict, optional) – Description

Returns textbook link.

Return type dict

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem () → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update ([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values () → list of D's values

viewitems () → a set-like object providing a view on D's items

viewkeys () → a set-like object providing a view on D's keys

viewvalues () → an object providing a view on D's values

Validator

exception `parsing.library.validator.`**MultipleDefinitionsWarning** (*data*, **args*)
Bases: *parsing.library.validator.ValidationWarning*

Duplicated key in data definition.

args

message

exception `parsing.library.validator.`**ValidationError** (*data*, **args*)
Bases: *parsing.library.exceptions.PipelineError*

Validator error class.

args

message

exception `parsing.library.validator.`**ValidationWarning** (*data*, **args*)
Bases: *parsing.library.exceptions.PipelineWarning*

Validator warning class.

args

message

class `parsing.library.validator.`**Validator** (*config*, *tracker=None*, *relative=True*)
Validation engine in parsing data pipeline.

config
`DotDict` – Loaded config.json.

course_code_regex
`re` – Regex to match course code.

kind_to_validation_function
`dict` – Map kind to validation function defined within this class.

KINDS
`set` – Kinds of objects that validator validates.

relative

`bool` – Enforce relative ordering in validation.

seen

`dict` – Running monitor of seen courses and sections

tracker

parsing.library.tracker.Tracker

KINDS = set(['textbook_link', 'datalist', 'meeting', 'section', 'textbook', 'course', 'config', 'eval', 'directory', 'instructor',

static **file_to_json**(*path*, *allow_duplicates=False*)

Load file pointed to by path into json object dictionary.

Parameters

- **path** (*str*) –
- **allow_duplicates** (*bool, optional*) – Allow duplicate keys in JSON.

Returns JSON-compliant dictionary.

Return type dict

classmethod **load_schemas**(*schema_path=None*)

Load JSON validation schemas.

NOTE: Will load schemas as static variable (i.e. once per definition), unless schema_path is specifically defined.

Parameters **schema_path** (*None, str, optional*) – Override default schema_path

static **schema_validate**(*data*, *schema*, *resolver=None*)

Validate data object with JSON schema alone.

Parameters

- **data** (*dict*) – Data object to validate.
- **schema** – JSON schema to validate against.
- **resolver** (*None, optional*) – JSON Schema reference resolution.

Raises `jsonschema.exceptions.ValidationError` – Invalid object.

validate(*data*, *transact=True*)

Validation entry/dispatcher.

Parameters **data** (*list, dict*) – Data to validate.

validate_course(*course*)

Validate course.

Parameters **course** (*DotDict*) – Course object to validate.

Raises

- *MultipleDefinitionsWarning* – Course has already been validated in same session.
- *ValidationError* – Invalid course.

validate_directory(*directory*)

Validate directory.

Parameters **directory** (*str, dict*) – Directory to validate. May be either path or object.

Raises *ValidationError* – encapsulated IOError

validate_eval(*course_eval*)
Validate evaluation object.

Parameters **course_eval** (DotDict) – Evaluation to validate.

Raises *ValidationError* – Invalid evaulation.

validate_final_exam(*final_exam*)
Validate final exam.

NOTE: currently unused.

Parameters **final_exam** (DotDict) – Final Exam object to validate.

Raises *ValidationError* – Invalid final exam.

validate_instructor(*instructor*)
Validate instructor object.

Parameters **instructor** (DotDict) – Instructor object to validate.

Raises *ValidationError* – Invalid instructor.

validate_location(*location*)
Validate location.

Parameters **location** (DotDict) – Location object to validate.

Raises *ValidationWarning* – Invalid location.

validate_meeting(*meeting*)
Validate meeting object.

Parameters **meeting** (DotDict) – Meeting object to validate.

Raises

- *ValidationError* – Invalid meeting.
- *ValidationWarning* – Description

validate_section(*section*)
Validate section object.

Parameters **section** (DotDict) – Section object to validate.

Raises

- *MultipleDefinitionsWarning* – Invalid section.
- *ValidationError* – Description

validate_self_contained(*data_path*, *break_on_error=True*, *break_on_warning=False*, *output_error=None*, *display_progress_bar=True*, *master_log_path=None*)
Validate JSON file as without ingestor.

Parameters

- **data_path** (*str*) – Path to data file.
- **break_on_error** (*bool, optional*) – Description
- **break_on_warning** (*bool, optional*) – Description
- **output_error** (*None, optional*) – Error output file path.

- **display_progress_bar** (*bool, optional*) – Description
- **master_log_path** (*None, optional*) – Description
- **break_on_error** –
- **break_on_warning** –
- **display_progress_bar** –

Raises *ValidationError* – Description

validate_textbook_link (*textbook_link*)
Validate textbook link.

Parameters textbook_link (DotDict) – Textbook link object to validate.

Raises *ValidationError* – Invalid textbook link.

validate_time_range (*start, end*)
Validate start time and end time.

There exists an unhandled case if the end time is midnight.

Parameters

- **start** (*str*) – Start time.
- **end** (*str*) – End time.

Raises *ValidationError* – Time range is invalid.

static validate_website (*url*)
Validate url by sending HEAD request and analyzing response.

Parameters url (*str*) – URL to validate.

Raises *ValidationError* – URL is invalid.

Logger

class parsing.library.logger.**JSONColoredFormatter** (*fmt=None, datefmt=None*)
Bases: logging.Formatter

converter ()

localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
tm_sec,tm_wday,tm_yday,tm_isdst)

Convert seconds since the Epoch to a time tuple expressing local time. When ‘seconds’ is not passed in, convert the current time instead.

format (*record*)

formatException (*ei*)
Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

formatTime (*record, datefmt=None*)
Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

usesTime()

Check if the format uses the creation time of the record.

class parsing.library.logger.**JSONFormatter**(*fmt=None*, *datefmt=None*)

Bases: logging.Formatter

Simple JSON extension of Python logging.Formatter.

converter()

localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
tm_sec,tm_wday,tm_yday,tm_isdst)

Convert seconds since the Epoch to a time tuple expressing local time. When 'seconds' is not passed in, convert the current time instead.

format(*record*)

Format record message.

Parameters **record** (*logging.LogRecord*) – Description

Returns Prettified JSON string.

Return type str

formatException(*ei*)

Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

formatTime(*record*, *datefmt=None*)

Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, the ISO8601 format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

usesTime()

Check if the format uses the creation time of the record.

class parsing.library.logger.**JSONStreamWriter**(*obj*, *type_=<type 'list'>*, *level=0*)

Bases: object

Context to stream JSON list to file.

BRACES

TYPE – Open close brace definitions.

file

dict – Current object being JSONified and streamed.

first

bool – Indicator if first write has been done by streamer.

level

int – Nesting level of streamer.

type_

dict, list – Actual type class of streamer (dict or list).

Examples

```
>>> with JSONStreamWriter(sys.stdout, type_=dict) as streamer:
...     streamer.write('a', 1)
...     streamer.write('b', 2)
...     streamer.write('c', 3)
{
    "a": 1,
    "b": 2,
    "c": 3
}
>>> with JSONStreamWriter(sys.stdout, type_=dict) as streamer:
...     streamer.write('a', 1)
...     with streamer.write('data', type_=list) as streamer2:
...         streamer2.write({0:0, 1:1, 2:2})
...         streamer2.write({3:3, 4:'4'})
...     streamer.write('b', 2)
{
    "a": 1,
    "data":
    [
        {
            0: 0,
            1: 1,
            2: 2
        },
        {
            3: 3,
            4: "4"
        }
    ],
    "b": 2
}
```

BRACES = {<type ‘dict’>: (‘{‘, ‘}’), <type ‘list’>: (‘[’, ‘]’)}

enter()

Wrapper for self.__enter__.

exit()

Wrapper for self.__exit__.

write(**args*, ***kwargs*)

Write to JSON in streaming fasion.

Picks either write_obj or write_key_value

Parameters

- ***args** – pass-through
- ****kwargs** – pass-through

Returns return value of appropriate write function.

Raises ValueError – **type_** is not of type list or dict.

write_key_value(*key*, *value=None*, *type_=<type 'list'>*)

Write key, value pair as string to file.

If value is not given, returns new list streamer.

Parameters

- **key** (*str*) – Description
- **value** (*str, dict, None, optional*) – Description
- **type** (*str, optional*) – Description

Returns None if value is given, else new JSONStreamWriter

write_obj(*obj*)

Write obj as JSON to file.

Parameters **obj** (*dict*) – Serializable obj to write to file.

parsing.library.logger.**colored_json**(*j*)

Tracker

class parsing.library.tracker.**NullTracker**(**args*, ***kwargs*)

Bases: *parsing.library.tracker.Tracker*

Dummy tracker used as an interface placeholder.

BROADCAST_TYPES = set(['TERM', 'STATS', 'YEAR', 'SCHOOL', 'MODE', 'TIME', 'DEPARTMENT', 'INSTRUCTO

add_viewer(*viewer*, *name=None*)

Add viewer to broadcast queue.

Parameters

- **viewer** (*Viewer*) – Viewer to add.
- **name** (*None, str, optional*) – Name the viewer.

broadcast(*broadcast_type*)

Do nothing.

department

end()

End tracker and report to viewers.

get_viewer(*name*)

Get viewer by name.

Will return arbitrary match if multiple viewers with same name exist.

Parameters **name** (*str*) – Viewer name to get.

Returns Viewer instance if found, else None

Return type *Viewer*

has_viewer(*name*)

Determine if name exists in viewers.

Parameters **name** (*str*) – The name to check against.

Returns True if name in viewers else False

Return type bool

instructor

mode

remove_viewer(*name*)

Remove all viewers that match name.

Parameters **name** (*str*) – Viewer name to remove.

report()

Do nothing.

school

start()

Start timer of tracker object.

stats

term

time

year

class parsing.library.tracker.**Tracker**

Bases: object

Tracks specified attributes and broadcasts to viewers.

@property attributes are defined for all BROADCAST_TYPES

BROADCAST_TYPES = set([’TERM’, ‘STATS’, ‘YEAR’, ‘SCHOOL’, ‘MODE’, ‘TIME’, ‘DEPARTMENT’, ‘INSTRUCTO

add_viewer(*viewer*, *name=None*)

Add viewer to broadcast queue.

Parameters

- **viewer** (*Viewer*) – Viewer to add.
- **name** (*None, str, optional*) – Name the viewer.

broadcast(*broadcast_type*)

Broadcast tracker update to viewers.

Parameters **broadcast_type** (*str*) – message to go along broadcast bus.

Raises *TrackerError* – if broadcast_type is not in BROADCAST_TYPE.

end()

End tracker and report to viewers.

get_viewer(*name*)

Get viewer by name.

Will return arbitrary match if multiple viewers with same name exist.

Parameters **name** (*str*) – Viewer name to get.

Returns Viewer instance if found, else None

Return type *Viewer*

has_viewer(*name*)
Determine if name exists in viewers.

Parameters **name** (*str*) – The name to check against.

Returns True if name in viewers else False

Return type bool

remove_viewer(*name*)
Remove all viewers that match name.

Parameters **name** (*str*) – Viewer name to remove.

report()
Notify viewers that tracker has ended.

start()
Start timer of tracker object.

exception parsing.library.tracker.**TrackerError**(*data*, **args*)
Bases: *parsing.library.exceptions.PipelineError*

Tracker error class.

args

message

Viewer

class parsing.library.viewer.**ETAProgressBar**
Bases: *parsing.library.viewer.Viewer*

receive(*tracker*, *broadcast_type*)

report(*tracker*)
Do nothing.

class parsing.library.viewer.**Hoarder**
Bases: *parsing.library.viewer.Viewer*

Accumulate a log of some properties of the tracker.

receive(*tracker*, *broadcast_type*)
Receive an update from a tracker.

Ignore all broadcasts that are not TIME.

Parameters

- **tracker** (parsing.library.tracker.Tracker) – Tracker receiving update from.
- **broadcast_type** (*str*) – Broadcast message from tracker.

report(*tracker*)
Do nothing.

schools
Get schools attribute (i.e. self.schools).

Returns Value of schools storage value.

Return type dict

class `parsing.library.viewer.`**`StatProgressBar`**(*stat_format='‘*, *statistics=None*)

Bases: *`parsing.library.viewer.Viewer`*

Command line progress bar viewer for data pipeline.

`SWITCH_SIZE` **= 100**

`receive`(*tracker*, *broadcast_type*)

Incremental update to progress bar.

`report`(*tracker*)

Do nothing.

class `parsing.library.viewer.`**`StatView`**

Bases: *`parsing.library.viewer.Viewer`*

Keeps view of statistics of objects processed pipeline.

`KINDS`

tuple – The kinds of objects that can be tracked. TODO - move this to a shared space w/Validator

`LABELS`

tuple – The status labels of objects that can be tracked.

`stats`

dict – The view itself of the stats.

`KINDS` **= (‘course’, ‘section’, ‘meeting’, ‘textbook’, ‘evaluation’, ‘offering’, ‘textbook_link’, ‘eval’)**

`LABELS` **= (‘valid’, ‘created’, ‘new’, ‘updated’, ‘total’)**

`receive`(*tracker*, *broadcast_type*)

Receive an update from a tracker.

Ignore all broadcasts that are not STATUS.

Parameters

- **`tracker`** (`parsing.library.tracker.Tracker`) – Tracker receiving update from.
- **`broadcast_type`** (*`str`*) – Broadcast message from tracker.

`report`(*tracker=None*)

Dump stats.

class `parsing.library.viewer.`**`TimeDistributionView`**

Bases: *`parsing.library.viewer.Viewer`*

Viewer to analyze time distribution.

Calculates granularity and holds report and 12, 24hr distribution.

`distribution`

dict – Contains counts of 12 and 24hr sightings.

`granularity`

int – Time granularity of viewed times.

`receive`(*tracker*, *broadcast_type*)

Receive an update from a tracker.

Ignore all broadcasts that are not TIME.

Parameters

- **tracker** (parsing.library.tracker.Tracker) – Tracker receiving update from.
- **broadcast_type** (*str*) – Broadcast message from tracker.

report (*tracker*)
Do nothing.

class parsing.library.viewer.**Timer** (*format='%(elapsed)s'*, ***kwargs*)
Bases: progressbar.widgets.FormatLabel, progressbar.widgets.TimeSensitiveWidgetBase

Custom timer created to take away 'Elapsed Time' string.

INTERVAL = datetime.timedelta(0, 0, 100000)

check_size (*progress*)

mapping = {u'seconds': (u'seconds_elapsed', None), u'max': (u'max_value', None), u'value': (u'value', None), u'elapsed

required_values = []

class parsing.library.viewer.**Viewer**
Bases: object

A view that is updated via a tracker object broadcast or report.

receive (*tracker*, *broadcast_type*)
Incremental updates of tracking info.

Parameters

- **tracker** (Tracker) – Tracker instance.
- **broadcast_type** (*str*) – Broadcast type emitted by tracker.

report (*tracker*)
Report all tracked info.

Parameters tracker (Tracker) – Tracker instance.

exception parsing.library.viewer.**ViewerError** (*data*, **args*)
Bases: *parsing.library.exceptions.PipelineError*

Viewer error class.

args

message

Digestor

class parsing.library.digestor.**Absorb** (*school*, *meta*)
Bases: *parsing.library.digestor.DigestionStrategy*

Load valid data into Django db.

meta
dict – Meta-information to use for DataUpdate object

school
str

classmethod digest_section (*parmams*, *clean=True*)

static remove_offerings(*section_obj*)

Remove all offerings associated with a section.

Parameters **section_obj** (`Section`) – Description

static remove_section(*section_code*, *course_obj*)

Remove section specified from database.

Parameters

- **section** (*dict*) – Description
- **course_obj** (`Course`) – Section part of this course.

wrap_up()

Update time updated for school at wrap_up of parse.

class parsing.library.digestor.**Burp**(*school*, *meta*, *output=None*)

Bases: *parsing.library.digestor.DigestionStrategy*

Load valid data into Django db and output diff between input and db data.

absorb

Vommit – Digestion strategy.

vommit

Absorb – Digestion strategy.

wrap_up()

class parsing.library.digestor.**DigestionAdapter**(*school*, *cached*)

Bases: `object`

Converts JSON defititions to model compliant dictionay.

cache

dict – Caches Django objects to avoid redundant queries.

school

str – School code.

adapt_course(*course*)

Adapt course for digestion.

Parameters **course** (*dict*) – course info

Returns Adapted course for django object.

Return type dict

Raises *DigestionError* – course is None

adapt_meeting(*meeting*, *section_model=None*)

Adapt meeting to Django model.

Parameters

- **meeting** (*TYPE*) – Description
- **section_model** (*None, optional*) – Description

Yields dict

Raises *DigestionError* – meeting is None.

adapt_section(*section*, *course_model=None*)

Adapt section to Django model.

Parameters

- **`section`** (*`TYPE`*) – Description
- **`course_model`** (*`None, optional`*) – Description

Returns formatted section dictionary

Return type dict

Raises *`DigestionError`* – Description

`adapt_textbook`(*textbook*)
Adapt textbook to model dictionary.

Parameters **`textbook`** (*`dict`*) – validated textbook.

Returns Description

Return type dict

`adapt_textbook_link`(*textbook_link*, *textbook_model=None*, *section_model=None*)
Adapt textbook link to model dictionary.

Parameters

- **`textbook_link`** (*`dict`*) – validated
- **`textbook_model`** (*`model, None, optional`*) –
- **`section_model`** (*`model, None, optional`*) –

Yields *dict* – model compliant

exception `parsing.library.digestor.`**`DigestionError`**(*data*, **args*)
Bases: *`parsing.library.exceptions.PipelineError`*

Digestor error class.

`args`

`message`

class `parsing.library.digestor.`**`DigestionStrategy`**
Bases: `object`

`wrap_up`()
Do whatever needs to be done to wrap_up digestion session.

class `parsing.library.digestor.`**`Digestor`**(*school*, *meta*, *tracker=<parsing.library.tracker.NullTracker object>*)
Bases: `object`

Digestor in data pipeline.

`adapter`
DigestionAdapter – Adapts

`cache`
dict – Caches recently used Django objects to be used as foriegn keys.

`data`
TYPE – The data to be digested.

`meta`
dict – meta data associated with input data.

MODELS
dict – mapping from object type to Django model class.

school
str – School to digest.

strategy
DigestionStrategy – Load and/or diff db depending on strategy

tracker
parsing.library.tracker.Tracker – Description

MODELS = {'textbook_link': <class 'timetable.models.TextbookLink'>, 'offering': <class 'timetable.models.Offering'>, 'se

digest (*data*, *diff=True*, *load=True*, *output=None*)
Digest data.

digest_course (*course*)
Create course in database from info in json model.

Returns django course model object

digest_meeting (*meeting*, *section_model=None*)
Create offering in database from info in model map.

Parameters **section_model** – JSON course model object

Return: Offerings as generator

digest_section (*section*, *course_model=None*)
Create section in database from info in model map.

Parameters **course_model** – django course model object

Keyword Arguments **clean** (*boolean*) – removes course offerings associated with section if set

Returns django section model object

digest_textbook (*textbook*)
Digest textbook.

Parameters **textbook** (*dict*) –

digest_textbook_link (*textbook_link*, *textbook_obj=None*, *section_obj=None*)
Digest textbook link.

Parameters

- **textbook_link** (*dict*) – Description
- **textbook_obj** (*Textbook, None, optional*) –
- **section_obj** (*Section, None, optional*) –

wrap_up ()

class parsing.library.digestor.**Vommit** (*output*)
Bases: *parsing.library.digestor.DigestionStrategy*

Output diff between input and db data.

diff (*kind*, *inmodel*, *dbmodel*, *hide_defaults=True*)
Create a diff between input and existing model.

Parameters

- **kind** (*str*) – kind of object to diff.
- **inmodel** (*model*) – Description
- **dbmodel** (*model*) – Description
- **hide_defaults** (*bool, optional*) – hide values that are defaulted into db

Returns Diff

Return type dict

static **get_model_defaults**()

remove_defaulted_keys(*kind, dct*)

wrap_up()

Exceptions

exception parsing.library.exceptions.**ParseError**(*data, *args*)

Bases: *parsing.library.exceptions.PipelineError*

Parser error class.

args

message

exception parsing.library.exceptions.**ParseJump**(*data, *args*)

Bases: *parsing.library.exceptions.PipelineWarning*

Parser exception used for control flow.

args

message

exception parsing.library.exceptions.**ParseWarning**(*data, *args*)

Bases: *parsing.library.exceptions.PipelineWarning*

Parser warning class.

args

message

exception parsing.library.exceptions.**PipelineError**(*data, *args*)

Bases: *parsing.library.exceptions.PipelineException*

Data-pipeline error class.

args

message

exception parsing.library.exceptions.**PipelineException**(*data, *args*)

Bases: exceptions.Exception

Data-pipeline exception class.

Should never be constructed directly. Use:

- PipelineError
- PipelineWarning

args

message

exception parsing.library.exceptions.**PipelineWarning** (*data*, **args*)

Bases: *parsing.library.exceptions.PipelineException*, exceptions.UserWarning

Data-pipeline warning class.

args

message

Extractor

class parsing.library.extractor.**Extraction** (*key*, *container*, *patterns*)

Bases: tuple

container

Alias for field number 1

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

key

Alias for field number 0

patterns

Alias for field number 2

parsing.library.extractor.**extract_info_from_text** (*text*, *inject=None*, *extractions=None*, *use_lowercase=True*, *splice_text=True*)

Attempt to extract info from text and put it into course object.

NOTE: Currently unstable and unused as it introduces too many bugs. Might reconsider for later use.

Parameters

- **text** (*str*) – text to attempt to extract information from
- **extractions** (*None, optional*) – Description
- **inject** (*None, optional*) – Description
- **use_lowercase** (*bool, optional*) – Description

Returns the text trimmed of extracted information

Return type str

Utils

class parsing.library.utils.**DotDict** (*dct*)

Bases: dict

Dot notation access for dictionary.

Supports set, get, and delete.

Examples

```
>>> d = DotDict({'a': 1, 'b': 2, 'c': {'ca': 31}})
>>> d.a, d.b
(1, 2)
>>> d['a']
1
>>> d['a'] = 3
>>> d.a, d['b']
(3, 2)
>>> d.c.ca, d.c['ca']
(31, 31)
```

as_dict()
Return pure dictionary representation of self.

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

has_key(*k*) → True if D has a key k, else False

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → list of D's values

viewitems() → a set-like object providing a view on D's items

viewkeys() → a set-like object providing a view on D's keys

viewvalues() → an object providing a view on D's values

class parsing.library.utils.**SimpleNamespace**(***kwargs*)

parsing.library.utils.**clean**(*dirt*)
Recursively clean json-like object.

***list*::**

- remove *None* elements

- *None* on empty list

`dict`::

- filter out None valued key, value pairs
- *None* on empty dict

***basestring*::**

- convert unicode whitespace to ascii
- strip extra whitespace
- None on empty string

Parameters **`dirt`** – the object to clean

Returns Cleaned *dict*, cleaned *list*, cleaned *string*, or pass-through.

`parsing.library.utils.`**`dict_filter_by_dict`**(*a*, *b*)

Filter dictionary a by b.

dict or set Items or keys must be string or regex. Filters at arbitrary depth with regex matching.

Parameters

- **a** (*dict*) – Dictionary to filter.
- **b** (*dict*) – Dictionary to filter by.

Returns Filtered dictionary

Return type dict

`parsing.library.utils.`**`dict_filter_by_list`**(*a*, *b*)

`parsing.library.utils.`**`dir_to_dict`**(*path*)

Recursively create nested dictionary representing directory contents.

Parameters **path** (*str*) – The path of the directory.

Returns Dictionary representation of the directory.

Return type dict

`parsing.library.utils.`**`iterrify`**(*x*)

Create iterable object if not already.

Will wrap *str* types in extra iterable eventhough *str* is iterable.

Examples

```
>>> for i in iterrify(1):
...     print(i)
1
>>> for i in iterrify([1]):
...     print(i)
1
>>> for i in iterrify('hello'):
...     print(i)
'hello'
```

`parsing.library.utils.`**`make_list`**`(`*x=None*`)`

Wrap in list if not list already.

If input is None, will return empty list.

Parameters **`x`** – Input.

Returns Input wrapped in list.

Return type list

`parsing.library.utils.`**`pretty_json`**`(`*obj*`)`

Prettify object as JSON.

Parameters **`obj`** (*`dict`*) – Serializable object to JSONify.

Returns Prettified JSON.

Return type str

`parsing.library.utils.`**`safe_cast`**`(`*val*, *to_type*, *default=None*`)`

Attempt to cast to specified type or return default.

Parameters

- **`val`** – Value to cast.
- **`to_type`** – Type to cast to.
- **`default`** (*`None, optional`*) – Description

Returns Description

Return type to_type

`parsing.library.utils.`**`time24`**`(`*time*`)`

Convert time to 24hr format.

Parameters **`time`** (*`str`*) – time in reasonable format

Returns 24hr time in format hh:mm

Return type str

Raises `ParseError` – Unparseable time input.

`parsing.library.utils.`**`titlize`**`(`*name*`)`

Format name into pretty title.

Will uppercase roman numerals. Will lowercase conjuctions and prepositions.

Examples

```
>>> titlize('BIOLOGY OF CANINES II')
Biology of Canines II
```

`parsing.library.utils.`**`update`**`(`*d*, *u*`)`

Recursive update to dictionary w/o overwriting upper levels.

Examples

```
>>> update({0: {1: 2, 3: 4}}, {1: 2, 0: {5: 6, 3: 7}})
{0: {1: 2}}
```

Parsing Models Documentation

class `parsing.models.`**`DataUpdate`**`(`**args*, ***kwargs*`)`

Stores the date/time that the school's data was last updated.

Scheduled updates occur when digestion into the database completes.

`school`

CharField – the school code that was updated (e.g. jhu)

`semester`

`ForeignKey` to `Semester` – the semester for the update

`last_updated`

DateTimeField – the datetime last updated

`reason`

CharField – the reason it was updated (default Scheduled Update)

`update_type`

CharField – which field was updated

`UPDATE_TYPE`

`tuple` of `tuple` – Update types allowed.

`COURSES`

str – Update type.

`EVALUATIONS`

str – Update type.

`MISCELLANEOUS`

str – Update type.

`TEXTBOOKS`

str – Update type.

Scheduled Tasks

Frontend Documentation

The Structure

Note: to understand the file structure, it is best to complete the following tutorial: EggHead Redux. We follow the same structure and conventions which are typical of React/Redux applications.

Our React/Redux frontend can be found in `static/js/redux` and has the following structure:

```
static/js/redux
- __fixtures__
- __test_utils__
- __tests__
- actions
- constants
- helpers
- init.jsx
- reducers
- ui
- util.jsx
```

Let's break down this file structure a bit by exploring what lives in each section.

`__fixtures__`: JSON fixtures used as props to components during tests.

`__test_utils__`: mocks and other utilities helpful for testing.

`__tests__`: unit tests, snapshot tests, all frontend driven tests.

`actions`: all Redux/Thunk actions dispatched by various components. More info on this (more info on this below: *Actions*)

`constants`: application-wide constant variables

`init.jsx`: handles application initialization. Handles flows (see *Flows Documentation*), the passing of initial data to the frontend, and on page load methods.

`reducers`: Redux state reducers. (To understand what part of state each reducer handles, see *Reducers*).

`ui`: all components and containers. (For more info see *What Components Live Where*).

`util.jsx`: utility functions useful to the entire application.

Init.jsx

This file is responsible for the initialization of the application. It creates a Redux store from the root reducer, then takes care of all initialization. Only in `init.jsx` do we reference JSON passed from the backend via `timetable.html`.

It is this JSON, called `initData` which we read into state as our initial state for the redux application. However, sometimes there are special *flows* that a user could follow that might change the initial state of the application at page load. For this we use flows which are documented more thoroughly at the following link: *Flows Documentation*.

Other actions required for page initialization are also dispatched from `init.jsx` including those which load cached timetables from the browser, alerts that show on page load, the loading of user's timetables if logged in, and the triggering of the user agreement modal when appropriate.

Finally, `init.jsx` renders `<SemesterlyContainer />` to the DOM. This is the root of the application.

Actions

The actions directory follows this structure:

```
static/js/redux/actions
- calendar_actions.jsx - exporting the calendar (ical, google)
- exam_actions.jsx - final exam scheduling/sharing
- modal_actions.jsx - openning/closing/manipulating all modals
- school_actions.jsx - getting school info
- search_actions.jsx - search/adv search
- timetable_actions.jsx - fetching/loading/manipulating timetables
- user_actions.jsx - user settings/friends/logged in functionality
```

Reducers

The reducers directory follows this structure:

```
static/js/redux/reducers
- alerts_reducer.jsx - visibility of alerts
- calendar_reducer.jsx
- classmates_reducer.jsx
- course_info_reducer.jsx
- course_sections_reducer.jsx
- custom_slots_reducer.jsx
- exploration_modal_reducer.jsx
- final_exams_modal_reducer.jsx
- friends_reducer.jsx
- integration_modal_reducer.jsx
- integrations_reducer.jsx
- notification_token_reducer.jsx
- optional_courses_reducer.jsx
- peer_modal_reducer.jsx
- preference_modal_reducer.jsx
- preferences_reducer.jsx
- root_reducer.jsx
- save_calendar_modal_reducer.jsx
- saving_timetable_reducer.jsx
- school_reducer.jsx
- search_results_reducer.jsx
- semester_reducer.jsx
- signup_modal_reducer.jsx
- terms_of_service_banner_reducer.jsx
- terms_of_service_modal_reducer.jsx
- textbook_modal_reducer.jsx
- timetables_reducer.jsx
- ui_reducer.jsx
- user_acquisition_modal_reducer.jsx
- user_info_reducer.jsx
```

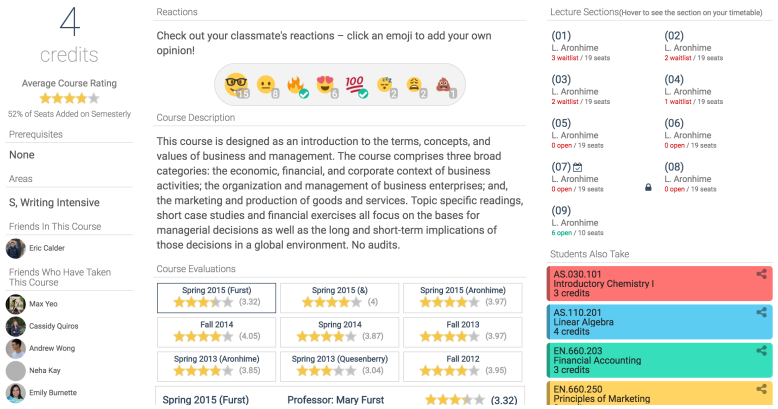
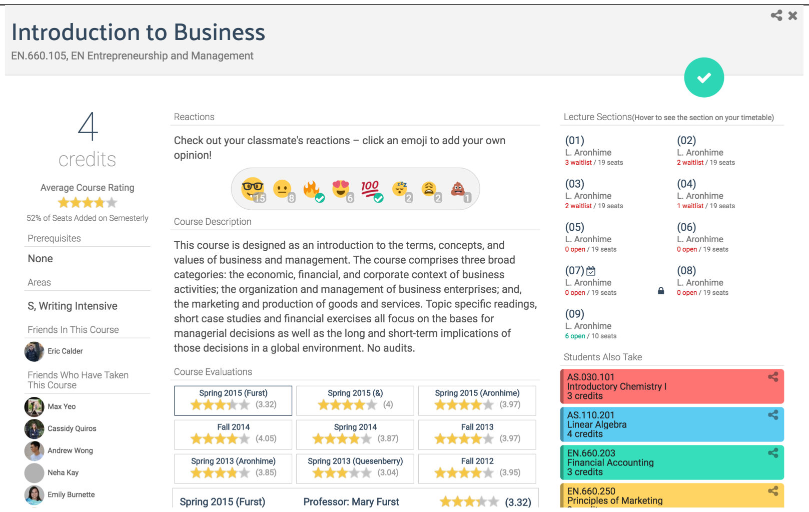
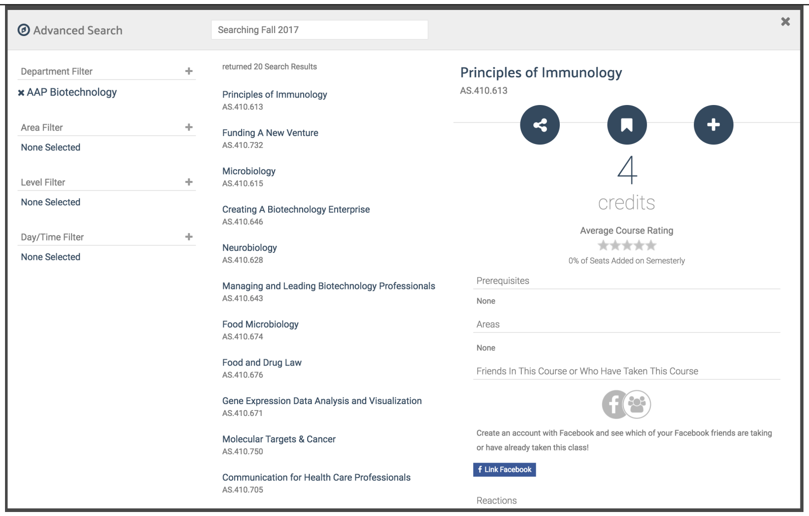
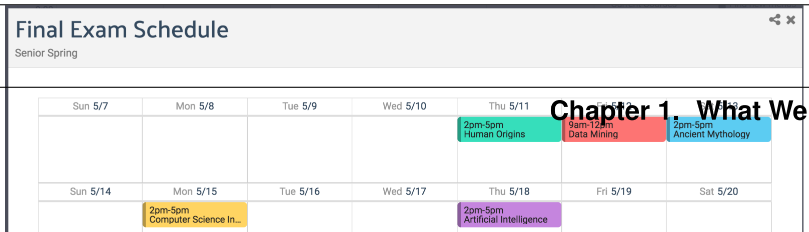
What Components Live Where

All of the components live under the `/ui` directory which follow the following structure:

```
static/js/redux/ui
- alerts
|   - ...
- containers
|   - ...
- modals
|   - ...
- ...
```

General components live directly under `/ui/` and their containers live under `/ui/contaners`. However alerts (those little popups that show up in the top right of the app), live under `/ui/alerts`, and all modals live under `/ui/modals`. Their containers live under their respective sub-directories.

Modals

Component File	Screenshot	Description
course_modal_body.jsx		
course_modal.jsx		
exploration_modal.jsx		
		

4
credits
Average Course Rating
52% of Seats Added on Semesterly
Prerequisites
None
Areas
S, Writing Intensive
Friends In This Course
Eric Calder
Friends Who Have Taken This Course
Max Yeo
Cassidy Quiros
Andrew Wong
Neha Kay
Emily Burnette

Reactions

Check out your classmate's reactions – click an emoji to add your own opinion!

Course Description

This course is designed as an introduction to the terms, concepts, and values of business and management. The course comprises three broad categories: the economic, financial, and corporate context of business activities; the organization and management of business enterprises; and, the marketing and production of goods and services. Topic specific readings, short case studies and financial exercises all focus on the bases for managerial decisions as well as the long and short-term implications of those decisions in a global environment. No audits.

Course Evaluations

Spring 2015 (Furst) (3.32)	Spring 2015 (&) (4)	Spring 2015 (Aronhime) (3.97)
Fall 2014 (4.05)	Spring 2014 (3.87)	Fall 2013 (3.97)
Spring 2013 (Aronhime) (3.85)	Spring 2013 (Quesenberry) (3.04)	Fall 2012 (3.95)

Spring 2015 (Furst) Professor: Mary Furst (3.32)

Lecture Sections(Hover to see the section on your timetable)

(01) L. Aronhime 3 waitlist / 19 seats	(02) L. Aronhime 2 waitlist / 19 seats
(03) L. Aronhime 2 waitlist / 19 seats	(04) L. Aronhime 1 waitlist / 19 seats
(05) L. Aronhime 0 open / 19 seats	(06) L. Aronhime 0 open / 19 seats
(07) L. Aronhime 0 open / 19 seats	(08) L. Aronhime 0 open / 19 seats
(09) L. Aronhime 6 open / 10 seats	

Students Also Take

AS.030.101
Introductory Chemistry I
3 credits

AS.110.201
Linear Algebra
4 credits

EN.660.203
Financial Accounting
3 credits

EN.660.250
Principles of Marketing

Introduction to Business
EN.660.105, EN Entrepreneurship and Management

4
credits
Average Course Rating
52% of Seats Added on Semesterly
Prerequisites
None
Areas
S, Writing Intensive
Friends In This Course
Eric Calder
Friends Who Have Taken This Course
Max Yeo
Cassidy Quiros
Andrew Wong
Neha Kay
Emily Burnette
Reactions
Check out your classmate's reactions – click an emoji to add your own opinion!
Course Description
This course is designed as an introduction to the terms, concepts, and values of business and management. The course comprises three broad categories: the economic, financial, and corporate context of business activities; the organization and management of business enterprises; and, the marketing and production of goods and services. Topic specific readings, short case studies and financial exercises all focus on the bases for managerial decisions as well as the long and short-term implications of those decisions in a global environment. No audits.
Course Evaluations
Spring 2015 (Furst) (3.32) Spring 2015 (&) (4) Spring 2015 (Aronhime) (3.97)
Fall 2014 (4.05) Spring 2014 (3.87) Fall 2013 (3.97)
Spring 2013 (Aronhime) (3.85) Spring 2013 (Quesenberry) (3.04) Fall 2012 (3.95)
Spring 2015 (Furst) Professor: Mary Furst (3.32)
Lecture Sections(Hover to see the section on your timetable)
(01) L. Aronhime 3 waitlist / 19 seats (02) L. Aronhime 2 waitlist / 19 seats
(03) L. Aronhime 2 waitlist / 19 seats (04) L. Aronhime 1 waitlist / 19 seats
(05) L. Aronhime 0 open / 19 seats (06) L. Aronhime 0 open / 19 seats
(07) L. Aronhime 0 open / 19 seats (08) L. Aronhime 0 open / 19 seats
(09) L. Aronhime 6 open / 10 seats
Students Also Take
AS.030.101 Introductory Chemistry I 3 credits
AS.110.201 Linear Algebra 4 credits
EN.660.203 Financial Accounting 3 credits
EN.660.250 Principles of Marketing

Advanced Search
Searching Fall 2017

Department Filter
AAP Biotechnology
Area Filter
None Selected
Level Filter
None Selected
Day/Time Filter
None Selected

returned 20 Search Results

Principles of Immunology
AS.410.613

Funding A New Venture
AS.410.732

Microbiology
AS.410.615

Creating A Biotechnology Enterprise
AS.410.646

Neurobiology
AS.410.628

Managing and Leading Biotechnology Professionals
AS.410.643

Food Microbiology
AS.410.674

Food and Drug Law
AS.410.676

Gene Expression Data Analysis and Visualization
AS.410.671

Molecular Targets & Cancer
AS.410.750

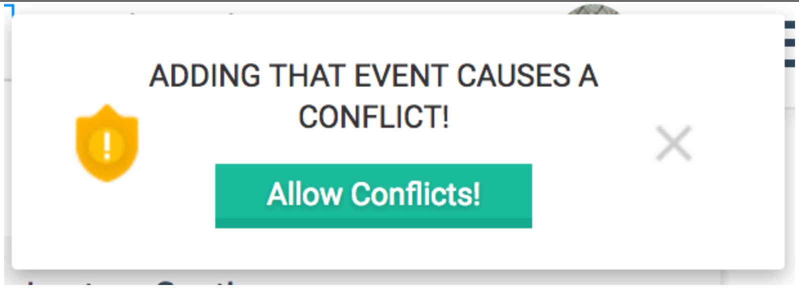
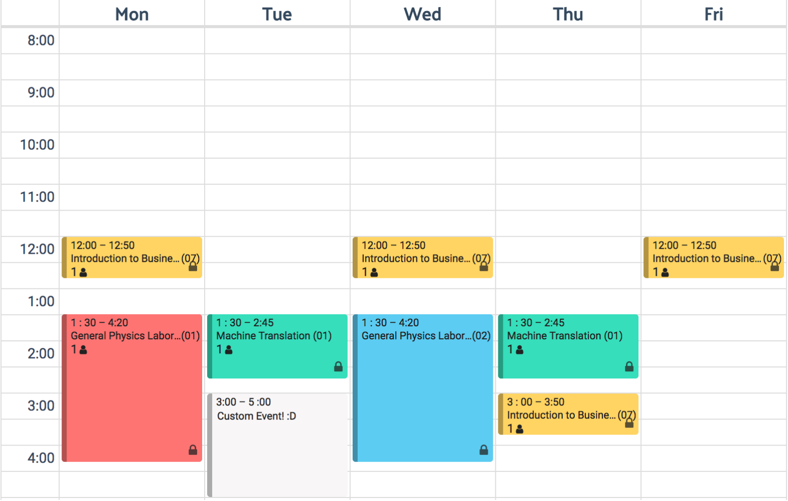
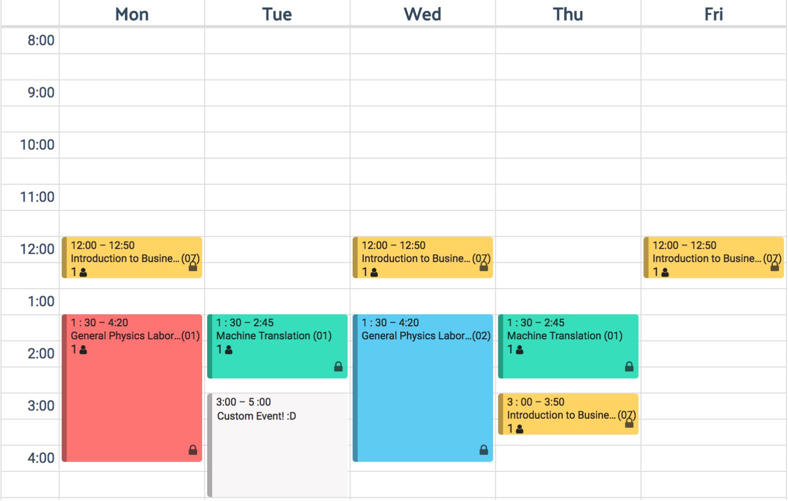
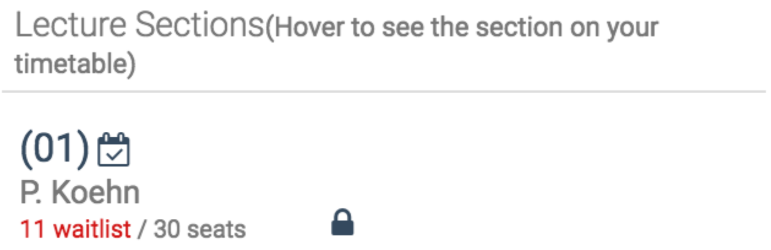
Communication for Health Care Professionals
AS.410.705

Principles of Immunology
AS.410.613
4
credits
Average Course Rating
0% of Seats Added on Semesterly
Prerequisites
None
Areas
None
Friends In This Course or Who Have Taken This Course
Create an account with Facebook and see which of your Facebook friends are taking or have already taken this class!
Link Facebook
Reactions

Final Exam Schedule
Senior Spring

Sun 5/7	Mon 5/8	Tue 5/9	Wed 5/10	Thu 5/11		
				2pm-5pm Human Origins	9am-12pm Data Mining	2pm-5pm Ancient Mythology
Sun 5/14	Mon 5/15	Tue 5/16	Wed 5/17	Thu 5/18	Fri 5/19	Sat 5/20
	2pm-5pm Computer Science In...			2pm-5pm Artificial Intelligence		

General Components

Component File	Screenshot	Description
alert.jsx	 ADDING THAT EVENT CAUSES A CONFLICT! Allow Conflicts!	
calendar.jsx	 Mon Tue Wed Thu Fri 8:00 9:00 10:00 11:00 12:00 1:00 2:00 3:00 4:00 12:00 – 12:50 Introduction to Busine...(07) 1 1 : 30 – 4:20 General Physics Labor...(01) 1 1 : 30 – 2:45 Machine Translation (01) 1 3:00 – 5 :00 Custom Event! :D 1 : 30 – 4:20 General Physics Labor...(02) 3 : 00 – 3:50 Introduction to Busine...(07) 1	
calendar.jsx	 Mon Tue Wed Thu Fri 8:00 9:00 10:00 11:00 12:00 1:00 2:00 3:00 4:00 12:00 – 12:50 Introduction to Busine...(07) 1 1 : 30 – 4:20 General Physics Labor...(01) 1 1 : 30 – 2:45 Machine Translation (01) 1 3:00 – 5 :00 Custom Event! :D 1 : 30 – 4:20 General Physics Labor...(02) 3 : 00 – 3:50 Introduction to Busine...(07) 1	
course_modal_sectio: jsx	 Lecture Sections(Hover to see the section on your timetable) (01) P. Koehn 11 waitlist / 30 seats	
	 9.00	

HTML/SCSS Documentation

Note: Although we write SCSS, you'll notice we use the SassLint tool and Sassloader. SASS is an older version of SCSS and SCSS still uses the old SASS compiler. Please don't write SASS, we're a SCSS shop. You can read about it briefly here.

What's in SCSS, What's not?

Written in SCSS:

1. Web Application

Written in plain CSS:

1. Splash pages
2. Pages for SEO
3. Emails and unsubscribe pages
4. basically everything that is not the web app

File Structure

All of our SCSS is in `static/css/timetable` and is broken down into 5 folders. The `main.scss` ties all the other SCSS files together importing them in the correct order.

Folder	Use
Base	`colors.scss` and `fonts.scss`
Vendors	any scss that came from a package that we wanted to customize heavily
Framework	`grid.scss` and `page_layout.scss`
Modules	styles for modular parts of our UI
Partials	component specific styles

All of the other CSS files in the `static/css` folder is either used for various purposes outlined above.

Linting and Codestyle

Note: Although we write SCSS, you'll notice we use the SassLint tool and Sassloader. SASS is an older version of SCSS and SCSS still uses the old SASS compiler. Please don't write SASS, we're a SCSS shop. You can read about it briefly here.

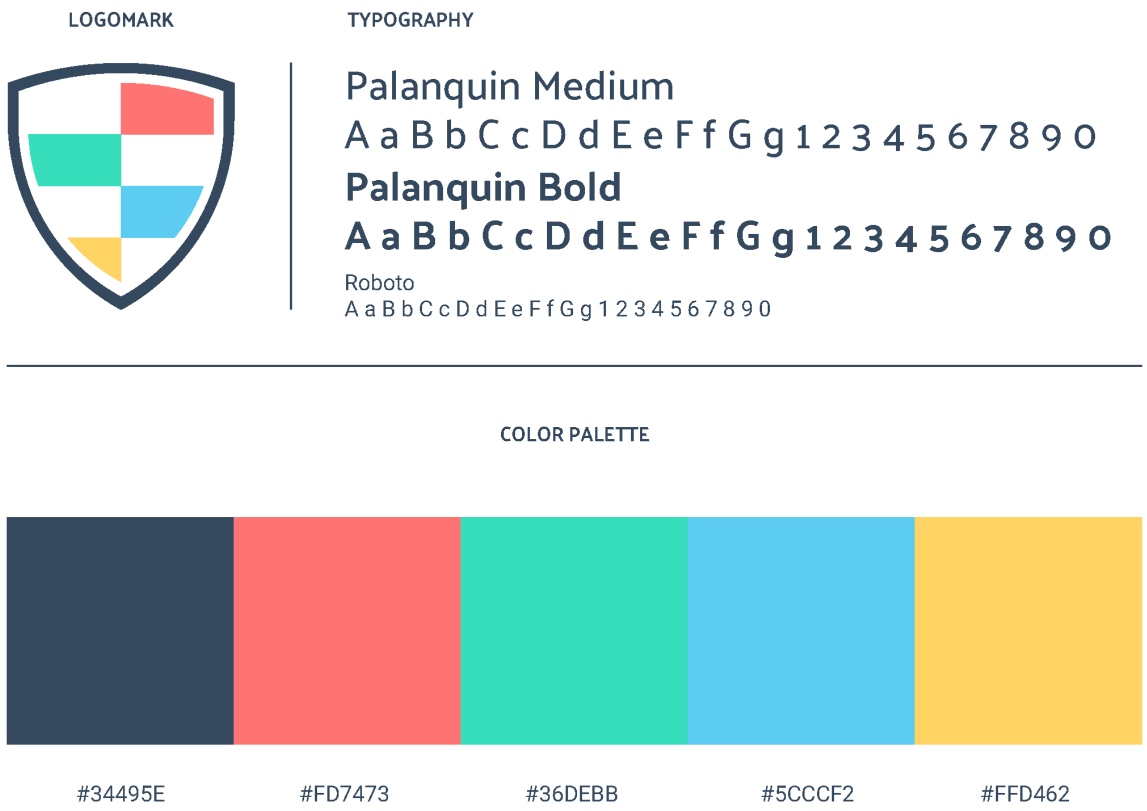
We use SASSLint with Airbnb's `.scss-lint.yml` file converted into `.sass-lint.yml`. Some things to take note of are

1. All colors must be declared as variables in `colors.scss`. Try your best to use the existing colors in that file
2. Double quotes
3. Keep nesting below 3 levels, use BEM
4. Use shortened property values when possible, i.e. `margin: 0 3px` instead of `margin: 0 3px 0 3px`
5. If a property is `0` don't specify units

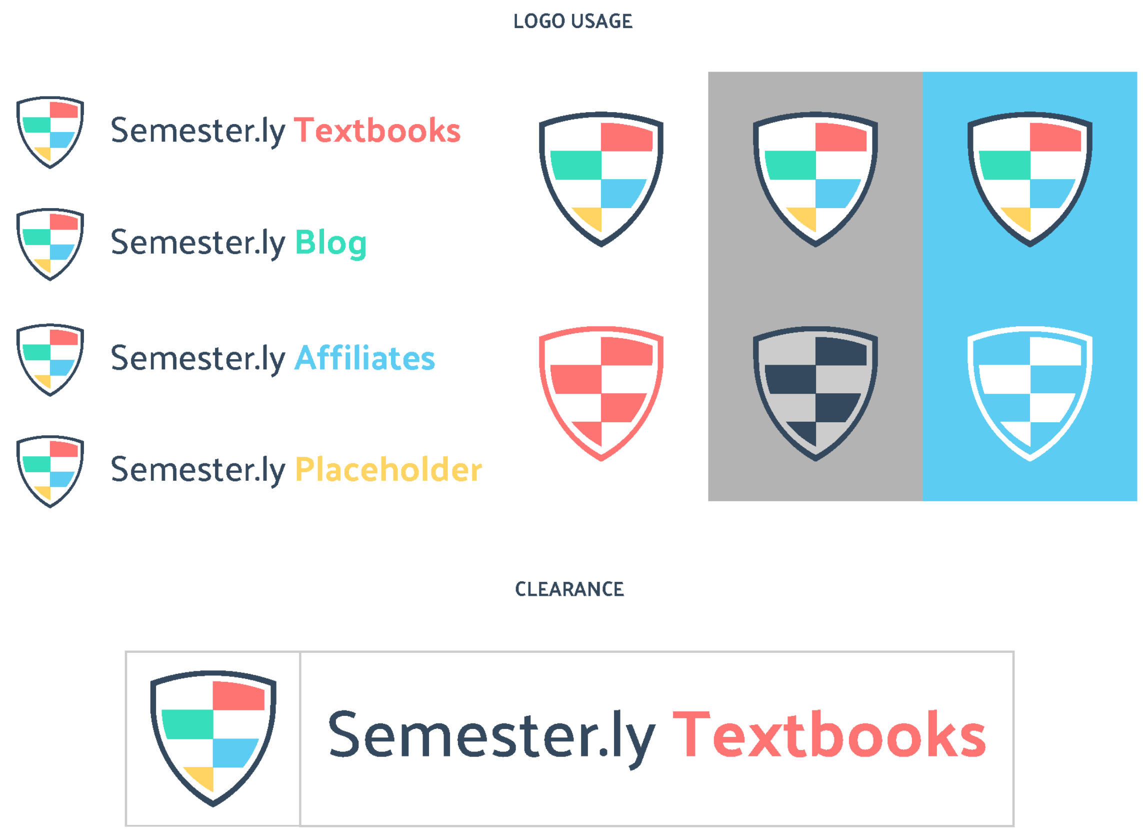
Refer to our `.sass-lint.yml` for more details and if you're using intelliJ or some IDE, use the sass-lint module to highlight code-style errors/warnings as you code.

Design/Branding Guidelines

Fonts & Colors


LOGOMARK
TYPOGRAPHY
Palanquin Medium
A a B b C c D d E e F f G g 1 2 3 4 5 6 7 8 9 0
Palanquin Bold
A a B b C c D d E e F f G g 1 2 3 4 5 6 7 8 9 0
Roboto
A a B b C c D d E e F f G g 1 2 3 4 5 6 7 8 9 0
COLOR PALETTE
#34495E
#FD7473
#36DEBB
#5CCCF2
#FFD462

Logo Usage



Logos

You can download logos and favicon files here

Editing This Documentation

Building the Docs

From the *docs* directory, execute the following command to rebuild all editted pages:

```
make html
```

To rebuild all pages, you may want to do a clean build:

```
make clean && make html
```

Viewing the Docs Locally

From the docs directory, open the index file from the build directory with the command:

```
open _build/html/index.html
```

Editing the Docs

All Django modules are documented via Sphinx AutoDoc. To edit this documentation, update the docstrings on the relevant functions/classes.

To update the handwritten docs, edit the relevant *.rst* files which are included by filename from *index.rst*.

Note: Be sure no warnings or errors are printed as output during the build process. Travis will build these docs and the build will fail on error.

Python Module Index

Index