Tangelo Programmer's Guide

In the following guide, the Tangelo team aims to familiarize new developers with our system, motivate major design choices, and describe Tangelo's system architecture at a high-level. We hope the document serves as a useful resource for our team and (hopefully) other future site maintainers, as well as any and all programmers looking to build something similar for themselves and their community. Please don't hesitate to reach out to our team if you experience any issues.

Please note, it may be difficult to view some of the diagrams in this document on smaller screens. If zooming in proves to be a problem, direct links to full resolution images are also prepended within each subsection.

Tangelo Start-Up

We expect the end-user (members of the Princeton community) to access the Tangelo platform deployed at Heroku (https://tangelo.herokuapp.com/). However, both for the current team and potential future maintenance programmers, it is useful to enumerate a few important steps required for local setup for future reference and platform development. Please see the readme for installation.

Tangelo System Architecture

Before we jump into the process flows of our system, we wanted to provide a brief summary of the early considerations that motivated the major design decisions behind our system.

Motivating the Front-End

Our main goals for Tangelo's UI are outlined here:

1. Minimal
2. Customizable
3. Engaging

To work toward achieving these initial goals, there were two major front-end system design decisions. Firstly, we wanted a self-contained dashboard. Users wouldn't have to leave their dashboard to access important functionalities, such as following new widgets or creating their own widget. Additionally, these features would be easily hidden from view using collapsible sidebars, such that the dashboard extends the full viewport.

On the topic of customizability and user widget interaction, Gridstack.js provided a quickstart in getting our first few widgets developed, with full drag and resizing functionality.

Motivating the Back-End

Our main goals for Tangelo's processing and data management layers are enumerated here:

1. Clean, expressive code
2. Secure
3. Fullstack python

Fortunately, a plethora of Flask extensions were available to meet these goals. Concerning our database, SQLAlchemy allowed us to treat all user data as objects for easy management. Flask's Jinja2 templating engine Jinja2 (combined with CSRF protection), promised security, and allowed for data to be passed smoothly from the database to the user. CAS was the right choice for our target users (the Princeton community). Combined with Flask_Login, the two offered an elegant solution for saving user sessions, and quickly accessing user information directly from our templates.

File Tree

The files most pertinent for a high-level understanding of Tangelo have the following tree structure. As indicated by an asterisk, start-up files (previously described) and less pertinent css and javascript files have been omitted for clarity.

```
.
├── tangelo
│   ├── CASClient.py
│   ├── __init__.py
│   ├── forms.py
│   ├── models.py
│   ├── models_service.py
│   ├── routes.py
│   ├── static
│   │   ├── dashboard.css
│   │   ├── grid.css
│   │   ├── tangelohome.css
│   │   └── *
│   ├── templates
│   │   ├── availableWidgets.html
│   │   ├── createWidget.html
│   │   ├── dashboard.html
│   │   ├── grid.html
│   │   ├── tangelohome.html
│   │   └── *
│   ├── user_utils.py
│   └── utils.py
└── *
```

User Login & Dashboard Accessibility

User Authentication: Flask-Login + CAS

Developed with the Princeton community in mind, CAS was an obvious choice for an easy, secure method of authenticating Tangelo users, both programmatically and for our users, who would be familiar with the process. We wanted quick access to the logged in user, and their data, for easy manipulation on the back-end, that would extend through to our Jinja templates on the front-end.

Flask-Login provided an interesting solution on top of CAS. All methods (excluding the home page landing route '/') are decorated with @login_required, for a clean and secure method of providing access to protected pages only to authenticated users. When a user attempts to access their dashboard without a prior login or saved user session, Flask automatically redirects them to Tangelo's /login view, which handles CAS authentication, and subsequently saves the user to Flask's user session.

All modules with app context have access to the Flask-Login current_user variable, which is an instance of the database User class. Before every request, Flask-Login will load the current_user. The tangelo.models.load_user() function, decorated with @login_manager.user_loader enables this process.

First Login: TigerBook API Request

CAS returns the netid for the authenticated user. Upon login, if a user does not exist in the database with the provided netid, a new User is created. On first login, a Tigerbook API request is made for that netid. If the netid is associated with an undergraduate student and the request is successful, the student's first name, last name, class year, and email are added to the newly created User. While these additional fields are not currently in use, they may be of use as we continue to develop Tangelo as a platform for the Princeton community.

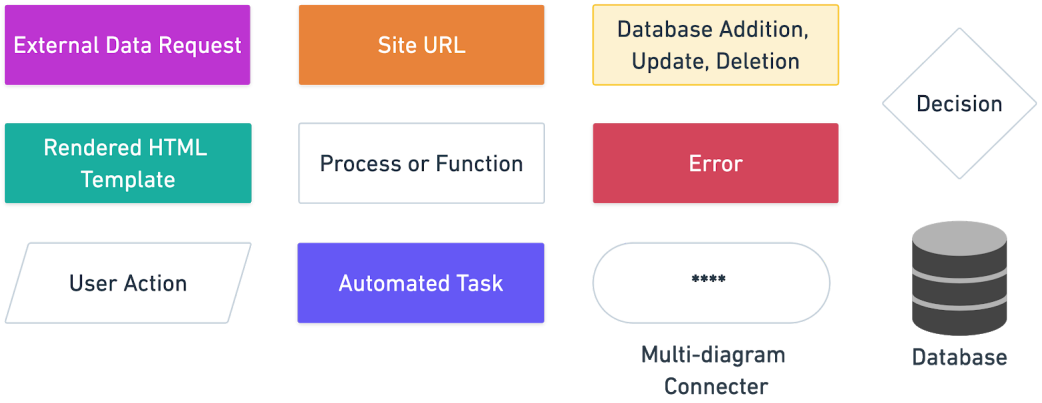
User Redirect

1. If a user is not logged in, and visits http://tangelo.herokuapp.com/, they are met with a clean, simple welcome page rendered by tangelohome.html.
2. If a user attempts to access '/dashboard' when not logged in, they are automatically directed to the CAS portal.
3. If a User is authenticated, they are always redirected to the dashboard view '/dashboard', rendered by dashboard.html.

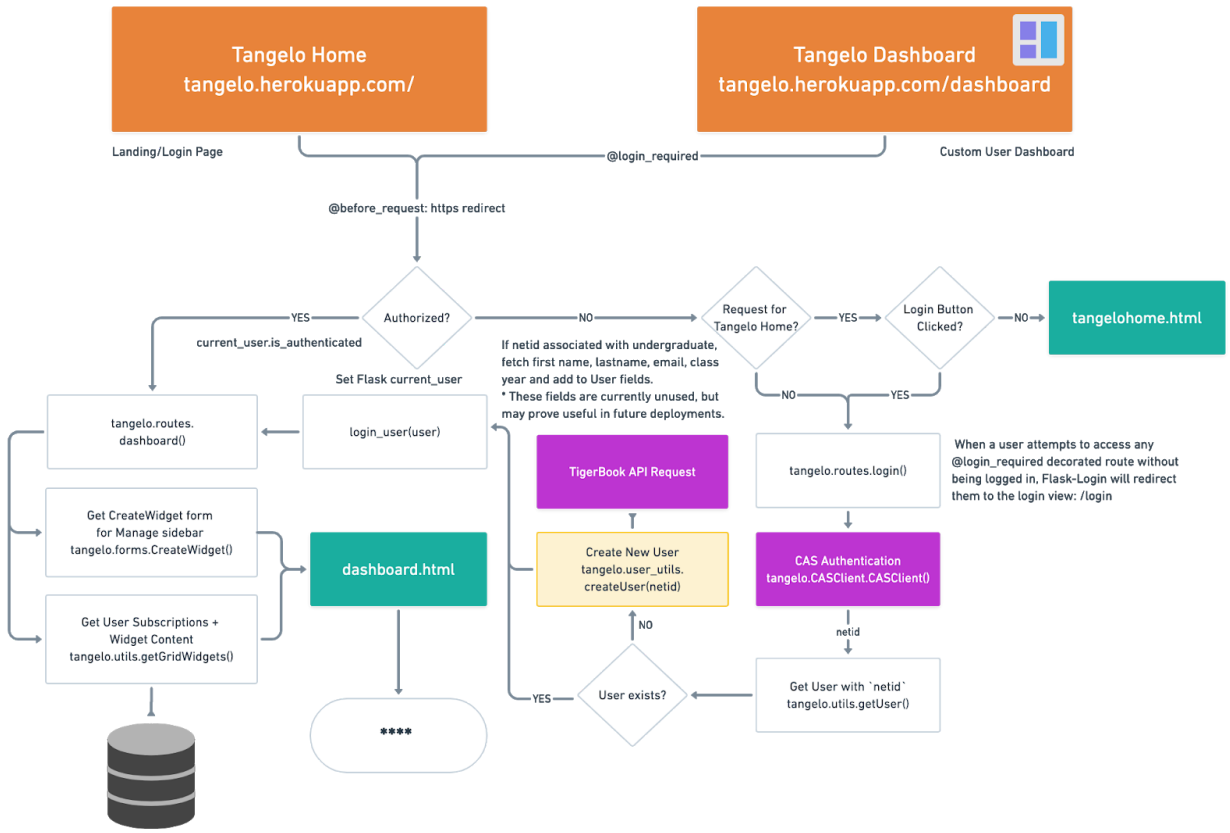
Authentication and Dashboard Display Diagram

The following diagram aims to provide additional support to the above in regards to the process of authenticating a user and displaying their dashboard.

Symbol Definitions



View Diagram in Full Resolution Here

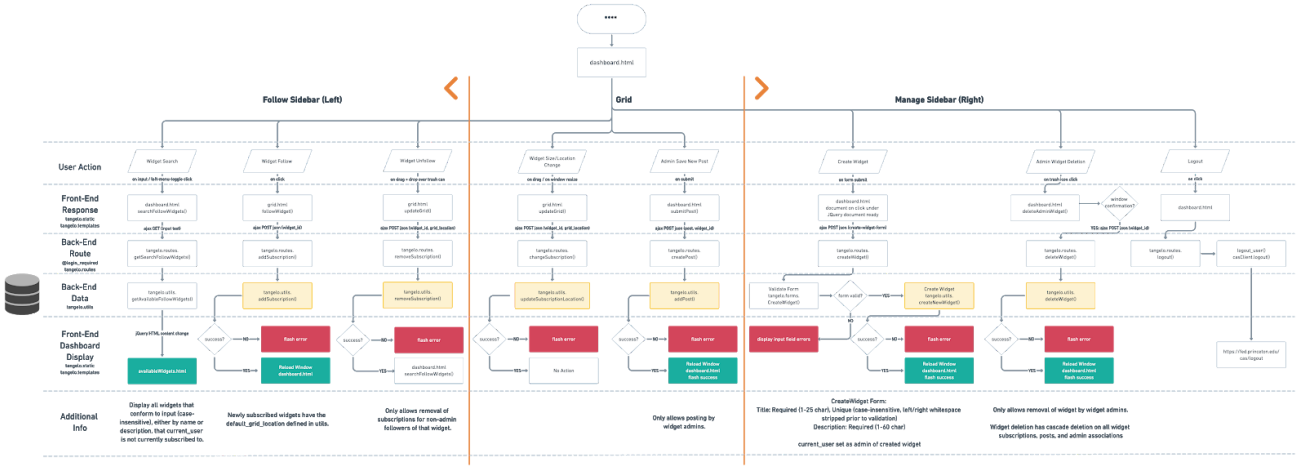


Dashboard User Actions: Fullstack Response Flow Diagram

The following diagram aims to show the full stack flow of our system, from user action to back-end requests, and from back-end data management to front-end response and display. The diagram is partitioned much like our front-end, with all logic associated with the left follow sidebar on the left, grid logic in the middle, and right manage sidebar logic on the right.

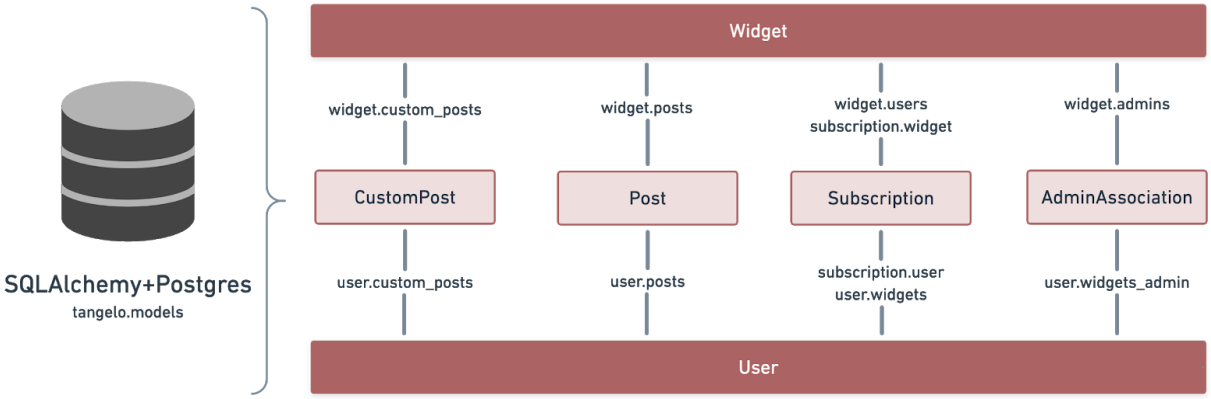
dashboard.html is our primary template, and includes the navigation bar, collapsible sidebars, and all necessary styling sheets and javascript functions. grid.html handles all grid logic, and is included as part of the main page content in dashboard.html as well.

View Diagram in Full Resolution Here



Database Schema

Tangelo's data is managed with SQLAlchemy with a Postgres database. This allows us to easily store, retrieve, manage, and manipulate the site's content through familiar Python object-oriented methods. Our two main classes are Widget and User. Between them are several association tables, namely CustomPost, Post, Subscription, and AdminAssociation, each having additional fields. These relationships, and methods of accessibility are shown here. For example, once a Widget object `example_widget` is returned from a database query, all subscribed users can be retrieved as simply as: `example_widget.users`. All database models are found in tangelo.models.py. The postgres database is handled on Heroku by the Heroku Postgres add-on.



The columns for each of these tables (classes) are described in detail here:

User (__tablename__ = 'users')				
Column	**Type**	**Unique?**	**Nullable?**	**Default Value?**
id (primary_key)	Integer	True	False	
netid	String	True	False	
email	String	True	True	
first_name	String	False	True	
last_name	String	False	True	
class_year	Integer	False	True	
birthday	DateTime	False	True	
create_dttm	DateTime	False	False	datetime.utcnow

Widget (__tablename__ = 'widgets')				
Field	**Type**	**Unique?**	**Nullable?**	**Default Value?**
id (primary_key)	Integer	True	False	
name	String	True	False	
alias_name	String	True	True	
description	String	False	False	
type	String	False	False	'generic' (alt: 'custom')
active	Boolean	False	False	False
post_limit	Integer	False	False	1
handle_display	String	False	False	'author' (alt: 'date_published')
style	String	False	True	
access_type *	String	False	False	'public' (alt: 'private')
post_type *	String	False	False	'private' (alt: 'public')
create_dttm	DateTime	False	False	datetime.utcnow

* These fields are not fully implemented, but represent future goals to make widgets more reflective of user feedback.

Subscription (__tablename__ = 'subscriptions')					
Field	**Type**	**Unique?**	**Nullable?**	**Default Value?**	**Foreign Key?**
id (primary_key)	Integer	True	False		
user_id	Integer	False	False		users.id
widget_id	Integer	False	False		widgets.id
grid_location	JSONEncodedDict	False	False		
create_dttm	DateTime	False	False	datetime.utcnow	

AdminAssociation (__tablename__ = 'administrators')					
Field	**Type**	**Unique?**	**Nullable?**	**Default Value?**	**Foreign Key?**
id (primary_key)	Integer	True	False		
user_id	Integer	False	False		users.id
widget_id	Integer	False	False		widgets.id
create_dttm	DateTime	False	False	datetime.utcnow	

Post (__tablename__ = 'posts')					
Field	**Type**	**Unique?**	**Nullable?**	**Default Value?**	**Foreign Key?**
id (primary_key)	Integer	True	False		
author_id	Integer	False	False		users.id
widget_id	Integer	False	False		widgets.id
content	String	False	False		
create_dttm	DateTime	False	False	datetime.utcnow	

CustomPost (__tablename__ = 'custom_posts')					
Field	**Type**	**Unique?**	**Nullable?**	**Default Value?**	**Foreign Key?**
id (primary_key)	Integer	True	False		
custom_author	Integer	False	True		
widget_id	Integer	False	False		widgets.id
content	String	False	False		
url	String	False	True		
create_dttm	DateTime	False	False	datetime.utcnow	

Database Unit Testing

Tangelo implements Python's unit testing framework to test our database table relationships and deletion cascades. These tests are described in the tests module in the root directory:

```
.
├── tests
│   ├── __init__.py
│   ├── test_db.py
```

We recommend all maintainers to read through the unit tests, as they may provide a good intuition as to how database objects can be created, committed, queried, updated, and deleted using SQLAlchemy and the provided database schema.

To run all tests sequentially:

$ python -m unittest tests.test_db

Tangelo Administered Widgets

Custom Widget Overview

Beyond the described Generic widget (allowing users to create their own widget, build a following, and generate their own content), our final COS333 submission of Tangelo includes 10 widgets designed by our team. These widgets are defined and added to the database upon setup (python ./setup_db.py). The relevant files for each are as follows:

Front-End Implementation:

1. Welcome to Tangelo (Demo)

```
.
├── tangelo
│   ├── static
│   │   ├── intro.css
│   │   ├── intro.js
│   │   ├── tangelo_intro.js
│   ├── templates
│   │   ├── welcomeWidget.html
```

2. Clock

```
.
├── tangelo
│   ├── static
│   │   ├── clock.css
```

```
│   │   ├── clock.js
│   ├── templates
│   │   ├── clock.html
```

3. Date

```
.
├── tangelo
│   ├── static
│   │   ├── date.css
│   │   ├── date.js
│   ├── templates
│   │   ├── date.html
```

4. Happy Birthday

```
.
├── tangelo
│   ├── static
│   │   ├── bcake.png
│   │   ├── dashboard.css
│   ├── templates
│   │   ├── birthday.html
```

5. Weather

```
.
├── tangelo
│   ├── static
│   │   ├── weather.css
│   ├── templates
│   │   ├── weather.html
```

Back-End Implementation:

6. Academic Calendar
7. COVID-19
8. News
9. Poem-a-Day
10. Princeton University News

```
.
├── tangelo
│   ├── static
│   │   ├── genericWidget.css
├── dataService
│   ├── __init__.py
│   ├── academic_calendar.py
│   ├── covid_api.py
│   ├── news_api.py
```

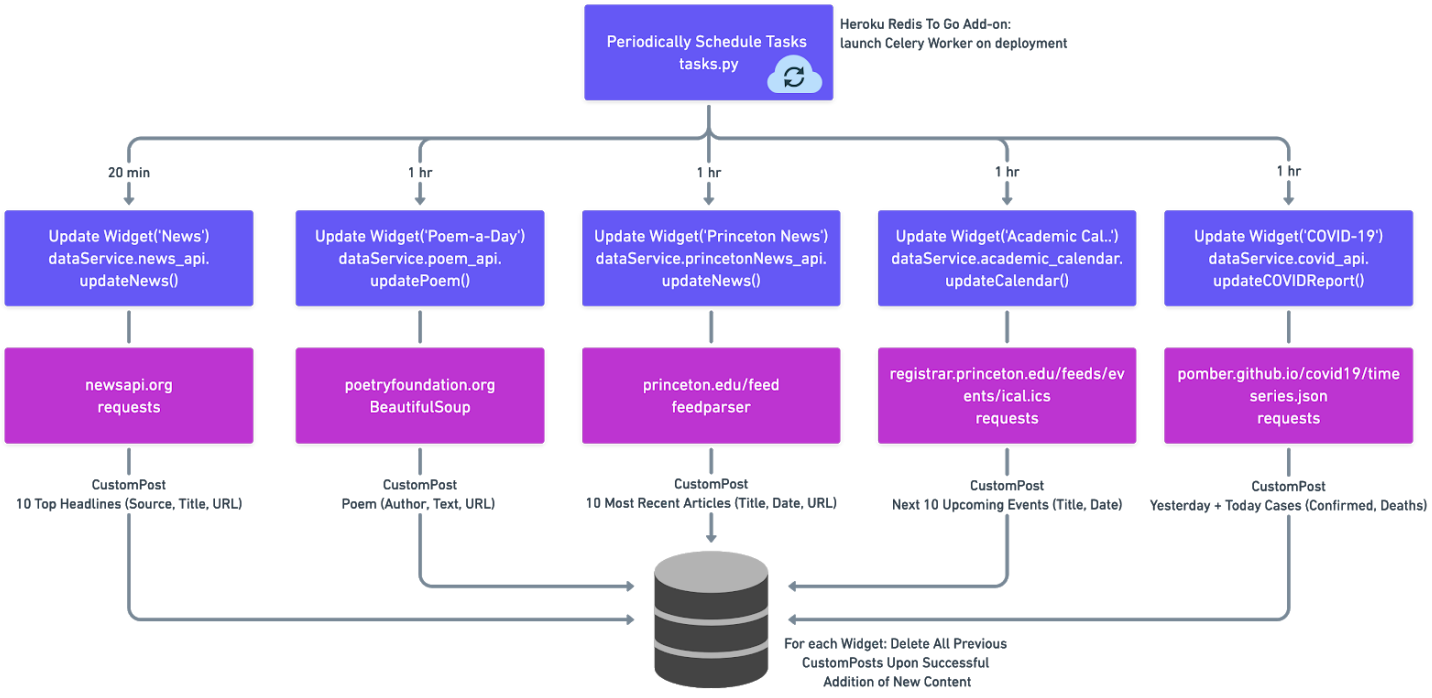
```
│   ├── poem_api.py
│   ├── princetonNews_api.py
├── setup_db.py
├── tasks.py
```

Automated Content Generation

5 of these 10 widgets, shown in purple, inherently require back-end support, as their displayed data is constantly evolving. Our goal was to display the most up-to-date information (current headlines, COVID-19 cases, Princeton calendar events, etc.) each time a user logs in to view their dashboard. To automate the process of updating tangelo administered content, we integrated the Celery Beat scheduler, allowing for script tasks to be executed at regular intervals (20-60 minutes) on Heroku. Each of these widgets is associated with a single script in the dataService module, that provides a method for requesting data from the relevant source, making the data beautiful, and updating the database. A single file in the tangelo directory (tasks.py) initializes the Celery worker and task schedule. All scheduled tasks are managed by a local Redis database. Heroku provides a Redis To Go Add-on, which sets the relevant environment variable REDISTOGO_URL. Additionally, the Procfile specifies the Celery worker.

A high-level overview of this process is shown below.

View Diagram in Full Resolution Here



Logging

Tangelo integrates Python's Logging framework. Our global logger `log` is initialized in tangelo.__init__.py. All modules in the tangelo package import this logger to display all messages at level INFO and above with ubiquitous, detailed formatting.

Security

All AJAX requests and FlaskForm submission are protected with a CSRF token using the Flask CSRFProtect extension. This registration is done within tangelo.__init__.py. HTML views are programmatically generated using the Jinja2 templating library. This is particularly important for displaying user generated content, as all data is automatically escaped when used in conjunction with Flask. We provide an additional layer of security on all back-end functions defined in tangelo.utils that:

- Confirm administrative roles of users when attempting to post a widget or delete a widget/subscription
- Validate data types and values passed by client requests

Design Problems

1. **Maintaining Custom Widget Templates.** Our initial approach to the problem of maintaining a diverse array of widget templates (and relevant CSS styles and Javascript script files) was to save the html for each custom widget as a string in the database. This was definitely not the best approach for several reasons:
    a. Custom widgets quickly became very difficult to maintain and update. Continuous updates to the widget template required continuous updates to the database.
    b. We ran into a severe problem with injections when testing the Create New Post functionality, as our approach also influenced a decision to move away from Jinja automatic escaping at a single, very critical point in the front-end.

    Fortunately, the injection attacks helped to guide a more sophisticated approach. The unique html template filename for each widget was saved in the database, and JQuery was used to load each template within the relevant widget container. This allowed us to easily update our templates on the fly, with no need to update the database.

2. **Dashboard Design.** Stylistically, we wanted a design language that maximized a user's interaction with the main dashboard; in order to achieve this, we decided that implementing the double sidebar best accommodated this goal while still allowing for dedicated sections for the widgets and settings. However, the sidebars' implementation led to some consequences that are difficult to adjust for. These include:
    a. The sidebar reduced the size of the dashboard. The opening and closing of the sidebar, particularly the left sidebar, would shift the horizontal boundaries of the dashboard in and out. This action would slow down the overall fluidity of the app, as all elements on the

screen were being resized each time. Further, the sidebar becomes incredibly intrusive on smaller screens, sometimes reducing the dashboard space by a third or more. On mobile interfaces, the dashboard is completely obstructed by the sidebars. This became especially pronounced when both sidebars were open.

b. The sidebars must be static. Sidebars do not scale content well, as they work with a predefined width. At some screen sizes, the dashboard can develop y-axis overflow, which will hide some needed functionality.

Some of these issues could be remedied by reordering/restyling the elements of the application such that sidebars expand over the dashboard, instead of next to. Another solution could be a pop-up, which is already within the design language with the posting feature.

3. **SQL Database.** For this project, given the timeline, course structure and support, and our limited prior familiarity with data management and related technologies, a SQL database with SQLAlchemy support was the ideal solution at the time. However, as we continue to build-out widget functionality, and diversify our outside data streams, one consideration that jumps to mind is the potential a NoSQL database may provide in terms of moving away from our current One-Table-Fits-All architecture. When we begin to consider the types of content that can be displayed in a widget, beyond a universal String data type, non-tabular relations would provide more flexibility in how we store, retrieve, manipulate, and display data for the end-user. Currently, this issue is alleviated through the CustomPost table, which allows for more unique data fields for our custom widgets, including the URL from which the post was sourced, the original author of the post content, and the original publish date of the post content.