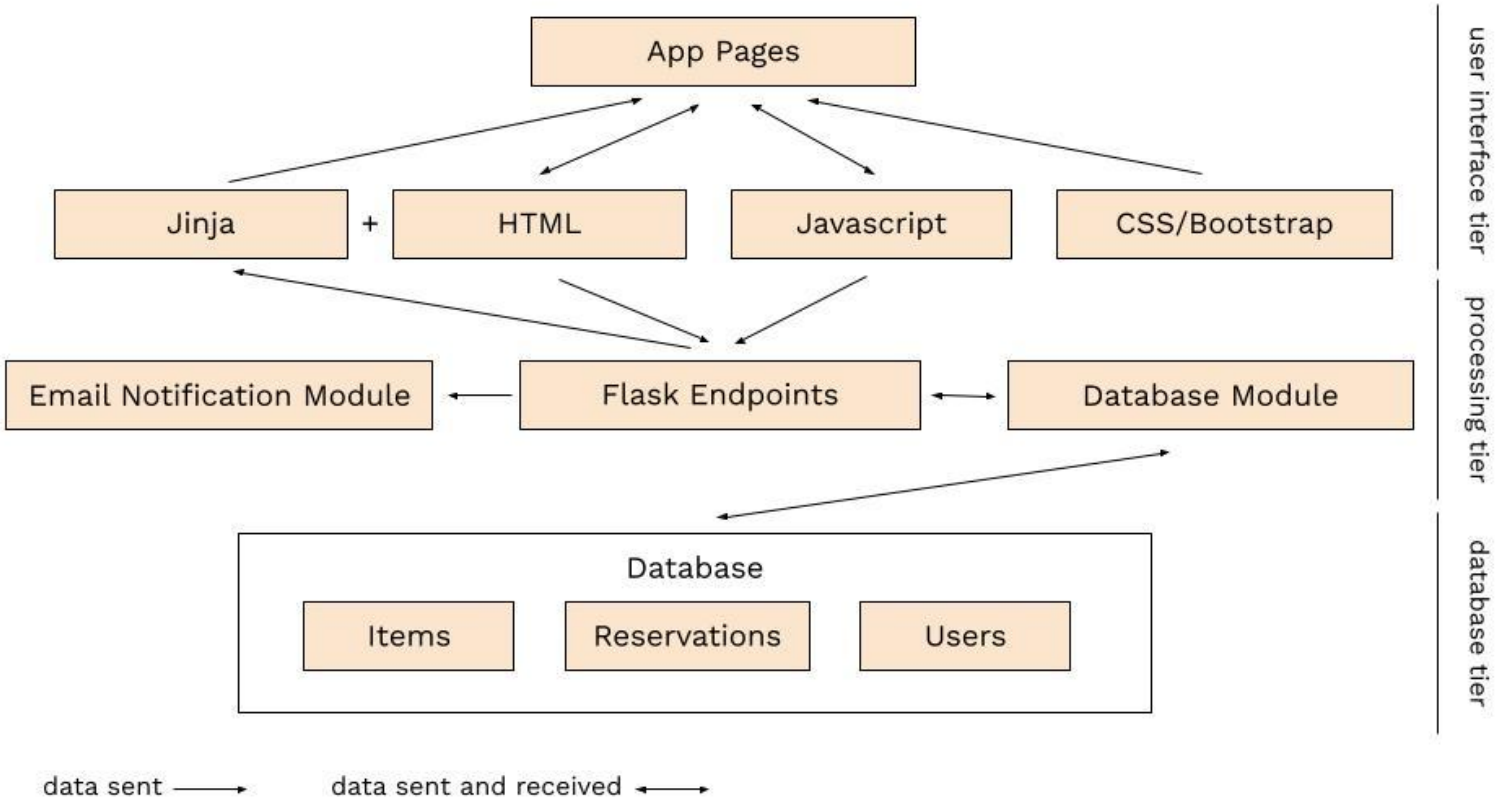
TigerThrift 

Programmer’s Guide

PROJECT ARCHITECTURE

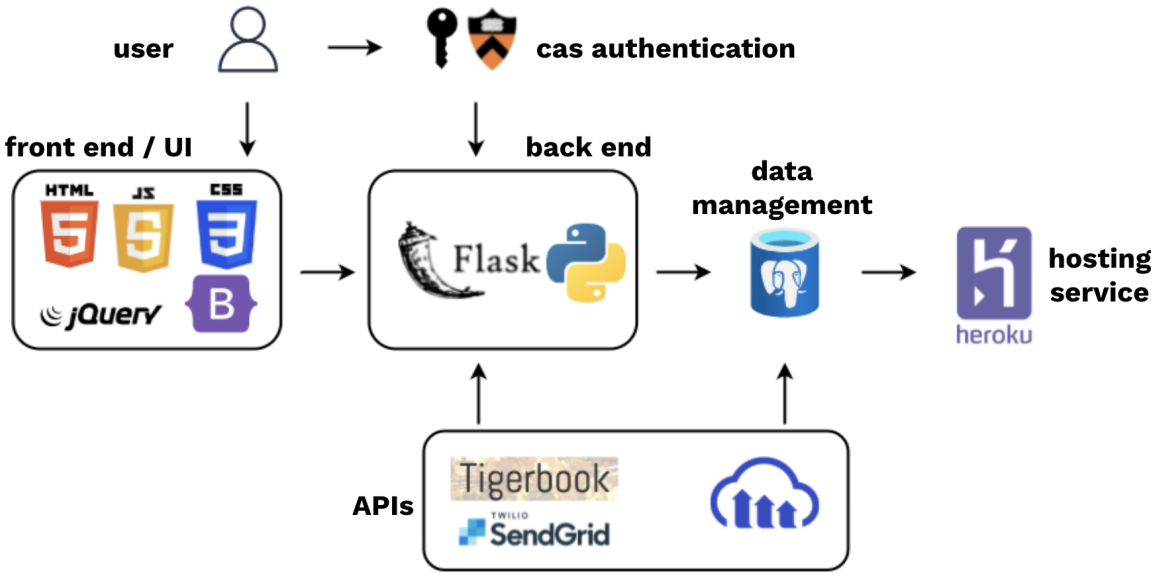


App pages are rendered by Jinja Templates, HTML, Javascript and Bootstrap, and these technologies and languages make up our user interface tier. Jinja templates create the app pages using data generated from the Flask endpoints as well as HTML files. Javascript also takes data from flask endpoints and displays it on app pages or uses it to determine what to display on app pages. CSS and Bootstrap style the pages.

Technologies used in our processing tier include Python and Flask. Our email notification module, our python file defining Flask endpoints, and our database module define our processing tier. The Flask endpoints file sends data to our user interface tier, and sometimes receives data in the form of a form submission from an HTML form. The Flask endpoints file sends data to the email notification module to send emails, and it accesses and changes data in the database. Not included in the diagram is our scheduled process that also connects to the database module and email notification mode, which is defined in more detail below.

Our database tier, a PostgreSQL database hosted by Heroku, is defined by three relational database tables that are changed and accessed by the database module.

Technologies used:



* We consider front end is synonymous with User Interface Tier and back end synonymous with Processing Tier

Our web-app is hosted by Heroku and accessible via https://tigerthrift.herokuapp.princeton.edu.

USER INTERFACE TIER

We have listed each of our pages, as well as the components and the details for each component in the table below. Though we did not include it in this table, CSS and Bootstrap are used on all of these pages for styling, design, and layout.

Page	Components	Details
/landing	HTML pages	Pages: landing.html - Places logo in center of page - Places brief description of our application - Creates button to log in (authenticate user)
	Jinja	n/a

	Javascript/ JQuery/AJAX	- Authenticates user with CasClient
/shop	HTML	Pages: header.html, shop.html, searchresults.html - Creates search bar - Creates filter select menu - Create sort by dropdown
	Jinja	- Includes header and search results html pages - Creates cards for each item with appropriate info and buttons for user - Remembers previous search/filter/sort
	Javascript/ JQuery/AJAX	- Generates search results on keystroke - Makes clear and apply filter buttons generate search results - Makes subtype/sizes filters change based on type selected
/sell	HTML pages	Pages: sell.html, header.html - Creates form to sell item
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	- Configures cloudinary upload widget - Creates delete button for uploaded images - Alerts to require at least one photo - Makes subtype/sizes select options change based on type selected
/about	HTML pages	Pages: about.html, header.html - Includes info about TigerThrift and links to contact us, as well as link to anonymous feedback Google form
	Jinja	- Includes header html page
	Javascript/ JQuery/AJAX	- Show different nav bar depending on whether user is logged in or not
/tutorial	HTML pages	Pages: tutorial.html, header.html - Creates tabs to explain the two main features (shop/sell)
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	- Show or hide information depending on which tab is being clicked - Show different nav bar depending on whether user is logged in or not

/profile	HTML pages	Pages: profile.html, header.html - Places information about user
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	- Reads in information about user
/mypurchased	HTML pages	Pages: mypurchased.html, header.html
	Jinja	- Includes header html pages - Creates card for each purchased item
	Javascript/ JQuery/AJAX	- Shows phone number if known
/myreserved	HTML pages	Pages: myreserved.html, header.html
	Jinja	- Includes header html pages - Creates card for each reserved item
	Javascript/ JQuery/AJAX	- Shows phone number if known
/mysellingactive	HTML pages	Pages: mysellingactive.html, header.html
	Jinja	- Includes header html pages - Creates card for each actively selling item
	Javascript/ JQuery/AJAX	- Shows phone number if known
/mysellingreserved	HTML pages	Pages: mysellingreserved.html, header.html
	Jinja	- Includes header html pages - Creates card for each reserved selling item
	Javascript/ JQuery/AJAX	- Shows phone number if known
/mysold	HTML pages	Pages: mysold.html, header.html
	Jinja	- Includes header html pages - Creates card for each sold item
	Javascript/ JQuery/AJAX	- Shows phone number if known
/itemdetails	HTML pages	Pages: itemdetails.html, header.html
	Jinja	- Includes header html pages

		- Shows correct buttons based on user relationship to item
	Javascript/ JQuery/AJAX	- Submits correct form for “go back” button, based on where the last visited page was
/edit	HTML pages	Pages: edit.html, header.html - Creates form to sell item - Buttons to save item, delete item, upload photo(s), go back to previous page
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	- Configures cloudinary upload widget - Creates delete button for uploaded images - Alerts to require at least one photo - Makes subtype/sizes select options change based on type selected
/cancelsuccess	HTML pages	Pages: success_cancel_ reservation.html, header.html - Shows success cancellation message and suggested next action buttons
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	n/a
/completesale	HTML pages	Pages: success_complete_reservation.html, header.html - Shows success sale completion message and suggested next action buttons
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	n/a
/editsuccess	HTML pages	Pages: success_edit.html, header.html
	Jinja	- Includes header html pages
	Javascript/ JQuery/AJAX	
/deletesuccess	HTML pages	Pages: success_item_deleted.html,header.html - Shows success deletion message and suggested next action buttons
	Jinja	- Includes header html page
	Javascript/	n/a

	JQuery/AJAX	
/reserve	HTML pages	Pages: success_reserve.html, header.html - Shows success reservation message and suggested next action buttons
	Jinja	- Includes header html page
	Javascript/ JQuery/AJAX	- Creates confirmation popup for cancel reservation button
/sellsuccess	HTML pages	Pages: success_sell.html, header.html - Show success in selling item message - Buttons to list another item, view items your selling, browse items
	Jinja	- Includes header html page
	Javascript/ JQuery/AJAX	- Submits correct form for “go back” button, based on where the last visited page was
/deletesuccess	HTML pages	Pages: success_item_deleted.html, header.html - Show success in deleting item message - Buttons to view items
	Jinja	- Includes header html page
	Javascript/ JQuery/AJAX	n/a
/error	HTML pages	Pages: error.html, header.html - Place error message - Buttons to browse items and contact us
	Jinja	- Includes header html page - Displays error message from processing tier
	Javascript/ JQuery/AJAX	n/a

HTML and Bootstrap / CSS

Our website was designed using HTML, CSS, and Bootstrap. HTML is used as the overall building blocks of the website, setting up the frame and structure of our website. We created a page (a .html file) for our main pages, as well as for certain portions of the website that was repeated throughout multiple pages – for example, ‘header.html’.

Within each of the pages, we used Bootstrap and pure CSS to design the pages to our liking. Bootstrap is a free, open-source tool that helps make the process of

making a responsive, mobile-friendly website. We used the latest Bootstrap version (5.1.2). Using Bootstrap allows us to get the general design of the website in place, such as creating a navbar and making it responsive to screen size, or adding flexing tags on certain divs so that the buttons respond to mobile screens. After formatting the different elements on the pages with Bootstrap, we went in with CSS to change the fonts, colors, and other design choices that we wanted to change from the pre-set designs that Bootstrap uses. This is a critical step, as this allows us to make our own, unique design choices – it adds our own touch.

Javascript / JQuery

We used Javascript as well, in script tags at the end of the body in most of our .html pages. To issue get requests, for example to load search results for the shop page (shop.html), we used AJAX and jQuery. To handle buttons and create onClick functions, we used javascript. Based on the relationship of a user to an item x (ex. If they're selling item x, or if they have item x reserved), the user has different capabilities: this is handled using javascript, to show/hide certain functionalities and buttons.

We also used Javascript to include our "Type" and "Subtype" / "Size" input form logic, where options for subtype and size update depending on "Type" selected.

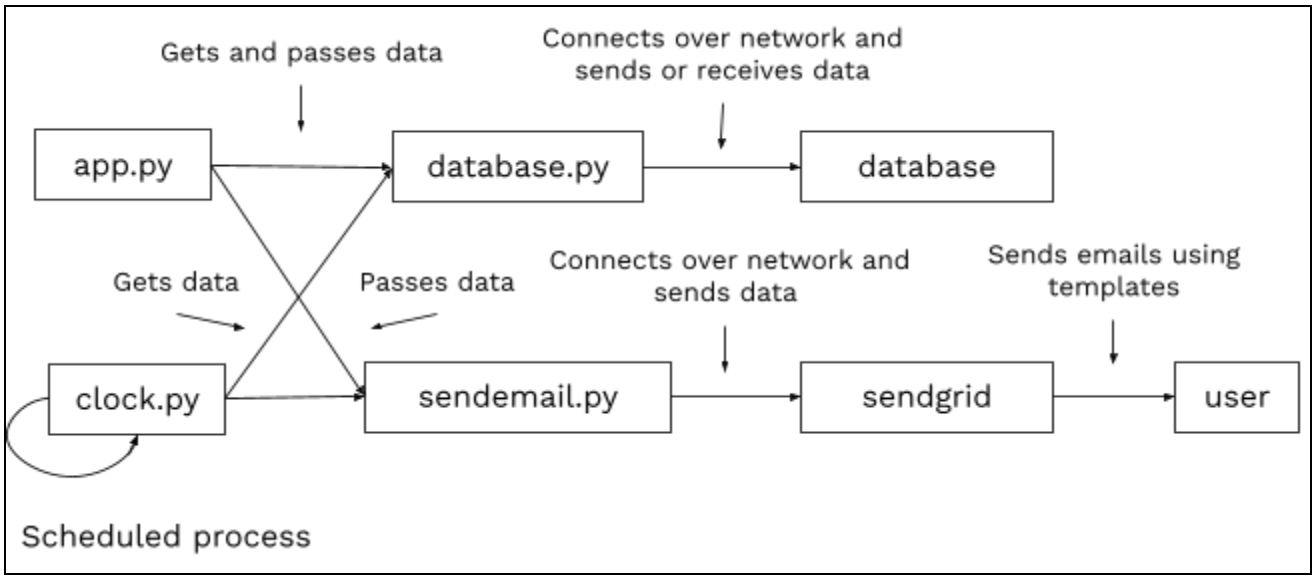
We used the most Javascript on our "Sell" page, to make the **cloudinary** upload widget, using cloudinary API, and to make delete buttons for images and accommodate multiple photo uploads and deletions.

For all delete/cancel buttons, which do riskier or more irreversible tasks, we added inline javascript in onSubmit attributes of html form elements to request confirmation using confirm().

We used jQuery to get each element by id, and add on click or on submit functions.

APPLICATION PROCESSING TIER

Technologies and languages used in our application processing tier include Flask and Python.



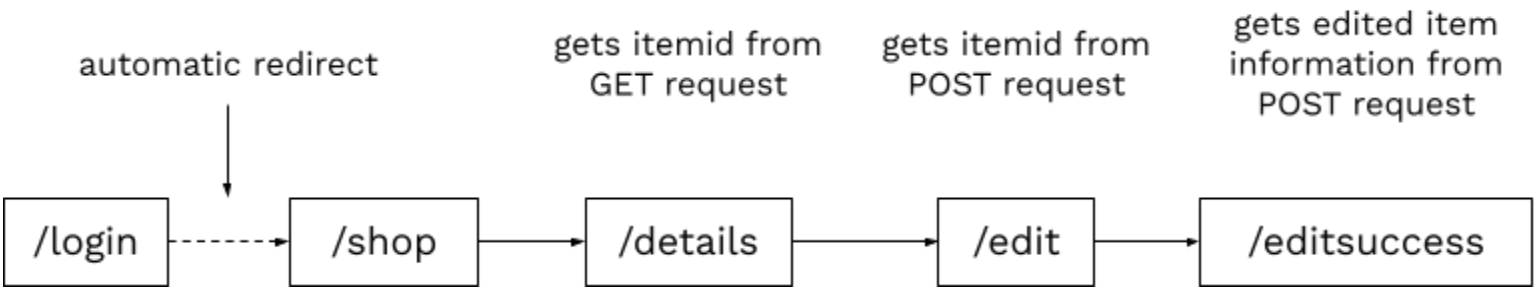
The processing tier consists of two main files: **app.py**, which defines all of the flask routes and **database.py** defines all of the functions used to access/query/update the database.

app.py

@app.route	Request Type	Description
/landing	GET	Checks if user has logged in via CAS, if so, redirects to /shop, if not, displays landing page
/login	GET	CAS authenticates users, redirects to /shop after successful authentication
/shop	GET	Authenticates user, generates shop page with previous search/sort/filter if applicable
/sell	GET	Authenticates user, then generates sell page with user info
/edit	POST	Authenticates user, then generates edit page with user info and stores previous search queries
/editsuccess	POST	Authenticates user, receives edited item information, edits item information in the database and generates a success page. Generates an error page if unsuccessful.
/sellsuccess	POST	Authenticates user, receives and stores selling item information in database, then generates a success page. Generates an error page if unsuccessful.

/searchresults	GET	Authenticates user, generates a page of item search results after searching with a given search/filter/sort. Generates an error page if search is unsuccessful.
/completesale	POST	Authenticates user, completes a sale and generates a success page. Generates an error page if unsuccessful.
/reserve	POST	Authenticates user, reserves an item, and generates a success page if successful. Generates an error page if unsuccessful.
/cancelsuccess	POST	Authenticates user, cancels a reservation and generates a success page if successful. Generates an error page if unsuccessful.
/deletesuccess	POST	Authenticates user, deletes item and generates a success page if successful. Generates an error page if unsuccessful
/profile	GET	Authenticates user, generates profile page with user info.
/mypurchased	GET	Authenticates user, generates page with a user's past purchased items. Generates error page if cannot get items.
/myreserved	GET	Authenticates user, generates page with a user's reserved items. Generates error page if cannot get items.
/myselling/active	GET	Authenticates user, generates page with a user's selling items that are not yet reserved. Generates error page if cannot get items.
/myselling /myselling/reserved	GET	Authenticates user, generates page with a user's selling items that are reserved by others. Generates error page if cannot get items.
/mysold	GET	Authenticates user, generates page with a user's previously sold items. Generates error page if cannot get items.
/itemdetails	GET	Authenticates user, checks if the item is sold or reserved by the user, and generates page with an item's details.
/about	GET	Generates about page (CAS authentication not required for access)
/tutorial	GET	Generates tutorial page (CAS authentication not required for access)
/logout	GET	Logs out user

Many routes that a user will take will include the generation of a page to do an action, followed by a success page that completes the action and indicates that it was successful. For an example, the user action to edit an item will use the following route:



This route is where the login will give the user access to the shop page. This will automatically redirect to /shop if a user successfully logs in, which then issues a get request to load to details page. The details page will then get the itemid from the get requests and render the details page. The edit page will get the itemid issued by a POST request from /details. Finally, the edited information from the edit page will then be POST requested to the /editsuccess page, and the /editsuccess page will get this information, make edits to the database, and then generate the success page.

The **database.py** file makes up all of the functions which interact with the database. The file includes various functions which each connect to the database and execute SQL statements to query or update the users, reservations, and items table in the database. Functions like reserving an item, editing an item, and selling an item are all completed with these database functions. Our database is configured using a DATABASE_URL environment variable defined in Heroku, and each function in the database.py file connects to the database via this URL.

sendemail.py is another file that makes up the processing tier, which consists of various functions to send emails using Sendgrid Email API. These functions are called when appropriate in **app.py**, and the scheduled process **clock.py.** They send emails with the sendgrid_api_key and email templates associated with our tigerthrift sendgrid account.

clock.py runs on a separate Heroku dyno as defined in our **Procfile**. This python script runs once a day at 4pm ET, and queries the database, using functions from our database module, for expired or about to expire reservations, and it calls the functions in **sendemail.py** that send the appropriate email reminders.

DATA MANAGEMENT TIER

We used three relational SQL databases to store information about our users, reservations, and items, the database is hosted by Heroku Postgres, and we used Cloudinary to store our images. Each item is given a unique itemid, which is how the items are identified throughout the database schema.

Items table:
Our items table includes information about items as well as links to the photos on Cloudinary. Most of this information is gathered from the sell form a user completes when they sell an item or edit an item. Other information, like status, is controlled by the developers (status of 0 means active, 1 is reserved, and 2 is inactive). Images are stored in Cloudinary when uploaded to the widget on the sell/edit pages where links to the photos are generated, and their links are stored in the items table.

itemid	prodname	type	subtype	size	gender	price	color	condition	brand	desc
181	Testing success sell	top	bodysuit	XS(W)	womens	$32.00	red	Brand New	leith	silk

photolink	photolink1	photolink2	posted	sellernetid	status	priceflexibility
http://res.cloudinarycom/...	http://res.cloudinary.com/...	http://res.cloudinary.com/...	2021-11-29 15:28:04	katelynr	2	price-negotiable

Reservations table:
Our reservations table stores information about a reservation made on item with itemid X, as well as the buyer, seller, and time of reservation. A row is created in the reservation table upon the reservation of an item, and when a reservation is complete the ‘completedtime’ column is filed.

itemid	buyernetid	sellernetid	reservedtime	completedtime
181	ishirai	katelynr	2021-11-30 22:55:36	2021-11-30 23:07:39

Users table:

Our users table stores information about users. A new row is generated after a user logs in for the first time (with information from TigerBook API). Phone number is added by the user optionally.

netid	email	joined	phone	first_name	last_name	full_name
ishirai	ishirai@princeton.edu	2021-10-31 22:50:39	6033066672	Iroha	Shirai	Iroha Shirai

Interesting Design Problems Encountered

Below, we have listed the design problems that we encountered and how we solved them.

1. **Bootstrap/CSS**

   We started off designing the website with just CSS, without using Bootstrap, but quickly realized that this was not a good idea, as using Bootstrap made the process of making our website mobile responsive to be much easier. However, there were trade offs with using Bootstrap; Bootstrap makes the formatting easier, but they have a lot of pre-set designs on how these elements (like buttons, nav-bars, headers) look like. In other words, Bootstrap puts limitations on the design of the website. In order to fix this design limitation, we decided to add a style.css file and made it so that it would overwrite the pre-set Bootstrap designs when we wanted it to; this allowed us to use the helpful features, like making the website mobile responsive, of Bootstrap, while still being expressive with our formatting and design of the website.

2. **Database Design**

   We decided to start off the project focusing on the database design, as we felt that our application was largely dependent on how we design and use the database that we created. Our database design was not a problem that we encountered after having created it; rather we saw it as something that could potentially create lots of problems, and thus our awareness of this allowed us to ‘solve’ this problem from the beginning. In doing so, we first began with a list of 1) all of the features we felt were critical to an item being sold, 2) the information about the actions (buying/selling) of an item, and 3) the information that we would need about the users of our application. Basing it off of these 3 main frameworks, we designed the database so that there was a

way that these 3 tables connected with each other. Ultimately, we resulted in 3 tables: items table, reservations table, and user table.

We believe that our approach in beginning with the database design, and spending a lot of time in doing so, was a successful step in our project, as this allowed us to not have to worry about the database in the later stages of the project. Furthermore, if necessary, it was easy to add different elements to the database given the well designed structure.

3. **Scheduler for Background Emails → AP Scheduler**

We had no problem setting up email notifications on a user's reservation, or on cancellation of a reservation, because upon such an action we could immediately call a function and send an email. We realized we wanted to send reminder emails about uncompleted reservations. We wanted to send emails that send on their own without a user completing an action, and before sending this email, check if the reservation had been completed, so as not to send an incorrect email. We then realized we needed to create a scheduled background process. We used the Advanced Python Scheduler (APScheduler) python library which allows us to write a python script that: 1) scans the reservations table for expired or near-expiring reservations, 2) sends appropriate emails, and 3) schedules how often this process runs. We scaled our process to a dyno and added .start() at the bottom of the file which begins the process. As it is configured in our Procfile, this is automatically called when it is deployed to Heroku. We deployed it to a different dyno (called clock) on Heroku such that it runs currently with the web app.