ShiftScheduler Programmer's Guide

Introduction to ShiftScheduler:
ShiftScheduler is a web-based platform for the Butler/Wilson dining hall employees of Princeton University to efficiently manage their shifts. We provide a wide variety of functionalities as well as an aesthetic interface.

In the dining hall student employment system, there are three roles--employee, manager, and coordinator. Everybody is an employee. Some employees are managers, and the main distinction is that they have a higher pay grade; in practice, they are basically the same. Finally, there is one employee who is a coordinator, and he/she coordinates all of the employees and shifts during the year.

In this document we will explain the internal structure of the ShiftScheduler and give some essential information that will help programmers as they carry out the maintenance of the application. We will start with the top-level components of the system and will go deeper into the details later in the document.

SECTION 1: TOP LEVEL

Figure 1, given below, describes the top level components of the ShiftScheduler and the communications between them. You can also see the names of all the files composing a component written inside the corresponding box. Below the figure, detailed explanations for all components and their communications are provided.

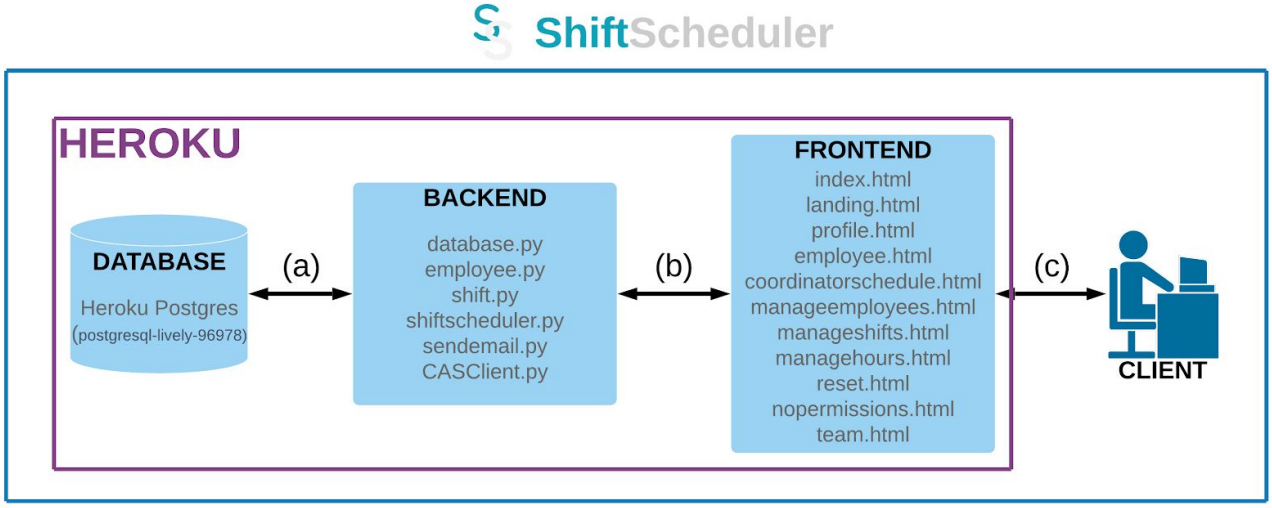


Figure 1: Top Level components of ShiftScheduler

Components:

1) **Heroku:** ShiftScheduler is deployed on Heroku as an application called “shiftscheduler”. Anyone that is added as a collaborator on Heroku can access the dashboard of ShiftScheduler by using the URL: https://dashboard.heroku.com/apps/shiftscheduler. Shiftscheduler is currently using a free Heroku plan under the “Free and Hobby” pricing category. The current deployment method of ShiftScheduler is configured to use the GitHub repository: https://github.com/aliekingurgen/shiftscheduler. “Procfile” and “requirements.txt” files that can be found in this repository are essential for successfully deploying to Heroku. If one desires to run ShiftScheduler on their local computer, they should also use the “requirements.txt” file as a reference. Note that as a result of this deployment method, a programmer who would like to contribute to ShiftScheduler should also be added as a collaborator in the GitHub repository. Please contact Ali Ekin Gurgen (agurgen@princeton.edu), if you would like to contribute to ShiftScheduler. Finally, the Heroku application of ShiftScheduler is configured to use a Heroku Postgres add-on, which will be explained with more details in the next part.

2) **Database:** ShiftScheduler’s database uses PostgreSQL as its relational database management system and it is deployed on Heroku Postgres. Heroku Postgres is an SQL database service provided by Heroku and it is attached as a “Database Add-On” to ShiftScheduler’s Heroku application. ShiftScheduler’s Heroku Postgres database is called “postgresql-lively-96978” and is under a free “Hobby-dev” pricing plan. The “database.py” file in the backend uses the URL of this Heroku Postgres database to create a connection. This URL and other information (i.e. Host, User, Password) about the database can be found under the “Settings” tab in the Heroku Postgres dashboard. The database will be explained in more detail in Section 2: Second Level.

3) **Backend:** The backend of ShiftScheduler uses Python, PostgreSQL (for sending SQL query statements to the database), and the Flask framework (for communicating with the frontend). The “database.py” file is the main module used for communicating with the database, and “shiftscheduler.py” file is the main module for communicating with the frontend. Other helper classes such as “employee.py” and “shift.py” are also essential components of the backend. We will explore all of the files that compose ShiftScheduler’s backend and the communication details between them in more detail in Section 2: Second Level.

4) **Frontend:** The frontend is built using HTML templates. We make the web application interactive using Javascript and JQuery. JavaScript scripts are used in every HTML page, and AJAX/JQuery in almost every page. In the setup() functions of each script we install event listeners to relevant components and if there is data to be loaded during setup, it’s also done in this function. Bootstrap is used for the formatting. In particular, Bootstrap modals and list groups are used heavily in our application.

5) **Client:** A client can write ShiftScheduler's public domain name in their browser (https://shiftscheduler.herokuapp.com/) to access the application's login page, where they will be required to go through Princeton CAS authentication to log in to the system. At this point a client can fall under 4 categories that are explained in the diagram below.

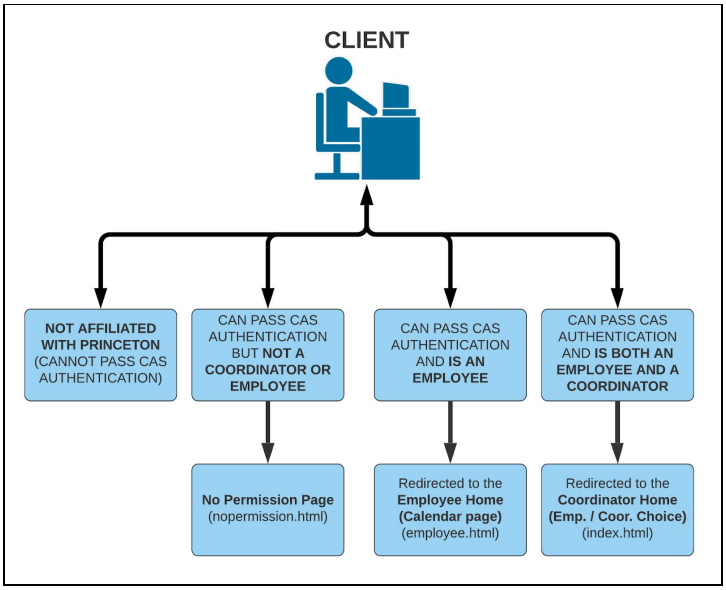


Figure 2: Possible Client Access Rights to ShiftScheduler

The client can then interact with the user interface in accordance with his/her access rights.

Communications:

1) **Database - Backend:** As mentioned earlier, "database.py" file in the backend uses the URL of the Heroku Postgres add-on to communicate with the database. Popular Python-PostgreSQL database adapter "psycopg2" module is used to create the connections.

2) **Backend - Frontend:**
The communication between the backend and the frontend is established through GET or POST requests. A lot of the pages on our application require communication with the backend while loading. For those pages Javascript's setup() function is used to establish a communication with the backend. Furthermore, the results are loaded into the web page dynamically using AJAX. JSON is sometimes used to transfer relevant data from the backend into the frontend.

3) **Frontend - Client:** ShiftScheduler is a web application. So the client can access ShiftScheduler by navigating to the URL: https://shiftscheduler.herokuapp.com/ in their favorite web browser. Although desktop is recommended, ShiftScheduler can be used both on mobile and desktop web browsers. The ShiftScheduler Team has used Chrome and Safari to test the application throughout the development process.

SECTION 2: SECOND LEVEL

1) **Database:** An entity-relationship diagram of the ShiftScheduler database is given below in Figure 2. The database consists of 9 tables as well as an additional 1-by-1 table (not shown in Figure 2) called current_pay_period, which is used for storing the beginning Monday of the current pay period. This “current_pay_period” table is incremented automatically by 14 days (since each pay period is 2 weeks) at the end of each pay period. Now we will explore the remaining 9 tables in more detail and explain how they interact with the methods defined in the “database.py” file in the backend.

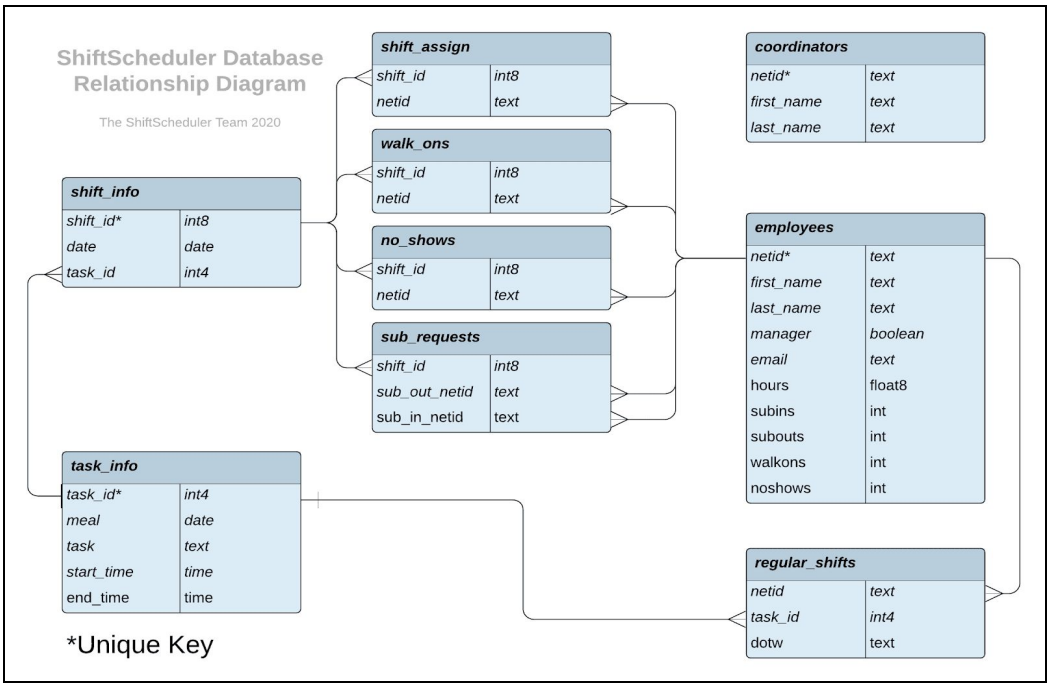


Figure 3: Entity-Relationship Diagram of the ShiftScheduler Database

- **coordinators:** This table stores the netid, first name and last name of the current coordinators in the dining hall. This is the table that the application admins, i.e. the ShiftScheduler Team, would have to manually edit in the backend in case there is a change of coordinators in the beginning of a semester. The “isCoordinator()” method is the only method in “database.py” that uses this table.

- **employees:** This table stores the netid, first name, last name, email, and a boolean indicating if they are a manager for all employees. This table also includes columns for storing some statistical data for each employee such as their worked hours in the current pay period, and their number of sub-ins, sub-outs, walk-ons and no-shows for the entire semester. The user interface does not allow you to remove yourself from the employees table and the system does not allow you to add an employee if the employee is already in the

employees table. In the table below, you can see a list of all methods in "database.py" (will be discussed in the next part) that can: (1) add/remove rows, (2) update rows, and (3) just query this table. Note that when removeEmployee is called, they are removed from the employees table but their shift information is still stored (but not used by the client unless they're re-added): this implementation was requested by Masi, the coordinator we worked with.

Can Add/Remove Rows	Can Update Rows	Can Only Query
insertEmployee() removeEmployee()	subOut() subIn() addWalkOn() addNoShow() undoNoShow() _hoursEmployee() resetStatsForEmployees()	isEmployee() employeeDetails() getAllEmployees() getEmployeeObject() getAllEmails() employeesInShift() walkOnsInShift()

- **task_info:** This is an immutable table that stores the information about each distinct 12 possible tasks in the dining hall (i.e. dinner manager, dinner first dish, brunch first shift, etc.). Each task is given an integer "task_id" between 1 - 12. Methods that can query this table are shiftDetails() and getTaskHours().

- **regular_shifts:** When a task_id (from the task_info table) is paired with a certain day of the week ("dotw"), this uniquely identifies a regular shift. This "task_id-dotw" pair is then assigned to a netid in the regular_shifts table. This table is used for getting an employee's regular shifts as well as for populating the shift_assign table. In other words, if there were no sub-ins, sub-outs, walk-ons or no-shows, the shift_assign table would essentially contain the same information as the regular_shifts table. However, when sub-ins, sub-outs, walk-ons and no-shows occur, the shift_assign table is modified but the regular_shifts table does not change. In the table below, you can see the methods that can modify this table. The addRegularShift() method checks for conflicting shifts (i.e. does not allow you to assign someone a Monday First Dinner if they are already assigned Monday Dinner Manager), and it prints a descriptive error message in the frontend if it catches any conflicts. Currently, conflicting shifts are checked using the _checkTaskConflicts method, which checks for conflicting tasks using a constant table stored in Python.

Can Add/Remove Rows	Can Update Rows	Can Only Query
addRegularShift() removeRegularShift()		regularShifts() populateForPeriod()

- **shift_info:** This table couples each date (in ISO format) and task_id pair and gives them a unique (auto-incremented) shift_id. This shift_id uniquely identifies any shift in the entire calendar and it is what we use for assigning shifts to netids in the shift_assign table. This table is populated using the populateForPeriod() method, which uses the task_info table to add rows between a given start and an end date. This populate method would usually be called once in the beginning of the year to populate shifts between September and May.

Can Add/Remove Rows	Can Update Rows	Can Only Query
populateShiftInfo()		shiftDetails() subOut() subIn() myShifts() addRegularShift() removeRegularShift() employeeObjectsInShift() addWalkOn() addNoShow() undoNoShow() getShiftHours() _hoursEmployee()

- **shift_assign:** This table is where the “shift_id - netid” pairs are being stored. When the shifts are being populated, populateForPeriod() method also populates the shift_assign table by using the current state of the regular_shifts table. Later, when there are changes in the shift assignments due to either (1) adding or removing regular shifts, or (2) sub-ins/sub-outs, walk-ons, no-shows occuring, the shift_assign table is modified as well. So in other words, at any given time, shift_assign table has the most recent information about the shifts an employee has worked in the past, and shifts they will work in the future. Therefore, this is the table we are using for getting an employee’s shifts for a week (for coloring them blue in the UI), and for calculating the hours they worked in the current pay period. The table below shows the methods that can interact with shift_assign.

Can Add/Remove Rows	Can Update Rows	Can Only Query
populateShiftInfo() addRegularShift() removeRegularShift() addWalkOn() addNoShow() undoNoShow()		allSubNeededForEmployee() myShifts() employeeObjectsInShift() _hoursEmployee()

- **walkons:** When an employee is added as a walk-on to a shift, the corresponding "shift_id - netid" pair is stored in this table. By having this as a separate table, we were able to keep the shift_assign table much simpler. The only method that can modify this table is addWalkOn(). The addWalkOn() method checks if the input netid is an Employee in the system. Similarly, addWalkOn() method checks for adding an employee as a walk-on to a shift more than once. It checks if the employee is already assigned to the shift that they are being added as a walk-on to. If any of these cases is caught, "Walk-On Request Failed" is printed in the frontend.

- **noshows:** When an employee is marked as no-show to a shift, the corresponding "shift_id - netid" pair is stored in this table. By having this as a separate table, we were able to keep the shift_assign table much simpler. The only methods that can modify this table are the addNoShow() and undoNoShow() methods.

- **sub_requests:** This table is the essential component of handling sub requests. When a sub is requested, in addition to removing the person that requested the sub from the shift_assign table, a row is being entered to the sub_requests table where the sub_in_netid column is "needed". As long as this column remains as "needed," this shift is being returned in the allSubNeededForEmployee() method and therefore be colored red in the UI. When someone takes a sub request, their netid replaces "needed" in the sub_in_netid column, and this shift is no longer being returned in the list of sub-needed shifts. Below you can see a more detailed list of all the methods that can interact with the sub_requests table. Sub-in and sub-out methods check if the shift that the employee is trying to sub-in/out from is from a past date. If it is, it gives an error message in the frontend.

Can Add/Remove Rows	Can Update Rows	Can Only Query
subOut() subIn()	subOut() subIn()	allSubNeeded() allSubNeededForEmployee()

2) **Backend:** The backend is composed of 6 main files and their relationships with each other can be seen in Figure 4 given below. We will now consider each component and their communications separately.

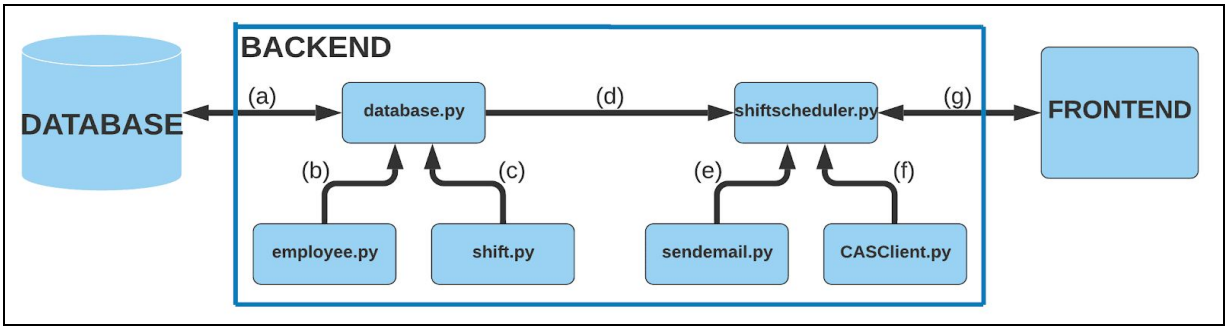


Figure 4: Components of the backend their communication relationships

- **database.py:** This file contains the definition of a class named Database. The Database class has a connect() method that should be called by the client (in this case shiftscheduler.py) before calling another database-related method, and a disconnect() method that should be called after the client is done. Other methods of the Database class can be used for performing the database-related operations. At the end of the file, you can see the unit testing for all methods which demonstrates how to call each one. You have already seen how these methods interact with the database tables in the previous part (see Section 2.1: Database). After explaining shiftscheduler.py, we will consider how the database methods interact with the functions of shiftscheduler.py.

- **employee.py:** This file contains the definition of a class named Employee. An Employee object contains exactly the same information as a row of the employees table, namely netid, first name, last name, email, manager, total hours, sub-ins, sub-outs, walk-ons, and no-shows. By using Employee objects, we were able to communicate the employee details much more easily between database.py and shiftscheduler.py. The employeeDetails(), getAllEmployees(), employeeObjectsInShift(), noShowsInShift(), and walkOnsInShift() methods create Employee objects based on the results of the database queries and return these Employee objects to their caller methods in shiftscheduler.py.

- **shift.py:** This file contains the definition of a class named Shift. A Shift object can be created by calling the shiftFromID() method in the database. It stores the shift_id, date, task_id, meal, task, start time, end time, and current number of workers. Given a unique shift_id, this method gets the relevant information from shift_info and task_info tables to create and return a Shift object. By using Shift objects, we were able to communicate Shift details much more easily between database.py and shiftscheduler.py. shiftDetails(), allSubNeeded(), and allSubNeededForEmployee() methods return Shift objects.

- **shiftscheduler.py:** shiftscheduler.py is the main file of our application. In order to run the application, one should run shiftscheduler.py with usage: "*python shiftscheduler.py port#*" or run the "runserver" file with usage: "*./runserver port#*". shiftscheduler.py handles the communication between database.py and the

frontend. It uses the Flask framework and [GET] and [POST] requests to handle the frontend communication. Each time it communicates with the database, it follows the following series of actions:

- Create a Database object -> *database = Database()*
- Connect to database -> *database.connect()*
- Call the needed database-related function -> e.g. *database.isCoordinator(netid)*
- Disconnect from database -> *database.disconnect()*

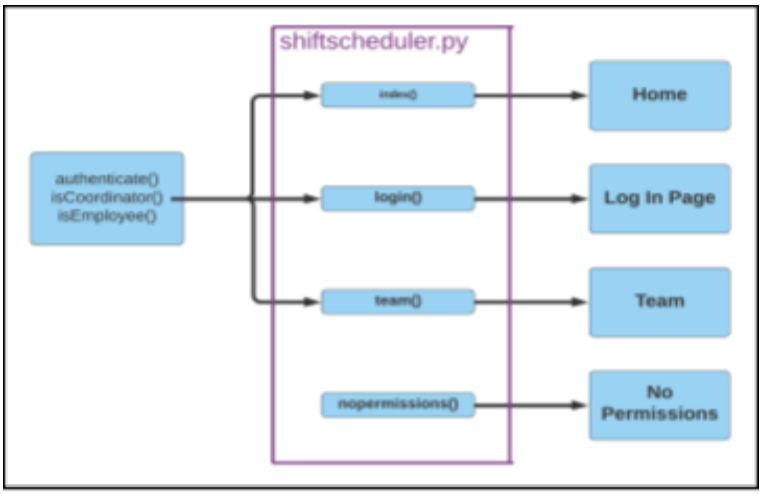


Figure 5: Communication between general frontend pages and backend

We will now focus on the functions in shiftscheduler.py and show how they create a communication between database and frontend. Figure 5 shows the communication between some general frontend pages (not specific to a role) on the right, and it shows the backend functions that are called by shiftscheduler.py on the left.

You will notice that almost all of the functions in shiftscheduler.py call the authenticate() method from CASClient() and isCoordinator() and isEmployee() methods from database.py. So for simplicity, we will not show these methods in the following diagrams. Next, we will look at the functions in shiftscheduler.py that create a communication between Employee frontend pages and related database functions.

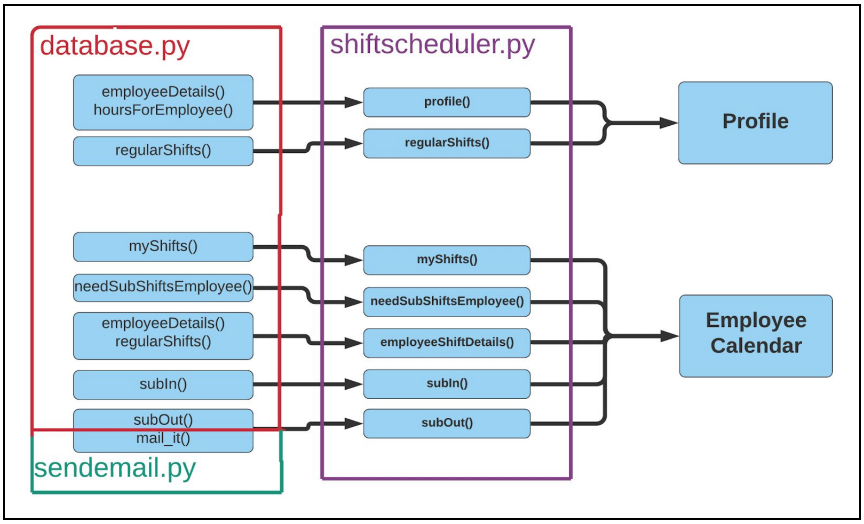


Figure 6: Communication between Employee frontend pages and backend

And finally we will take a look at the communication diagram between Coordinator frontend pages and the corresponding backend functions.

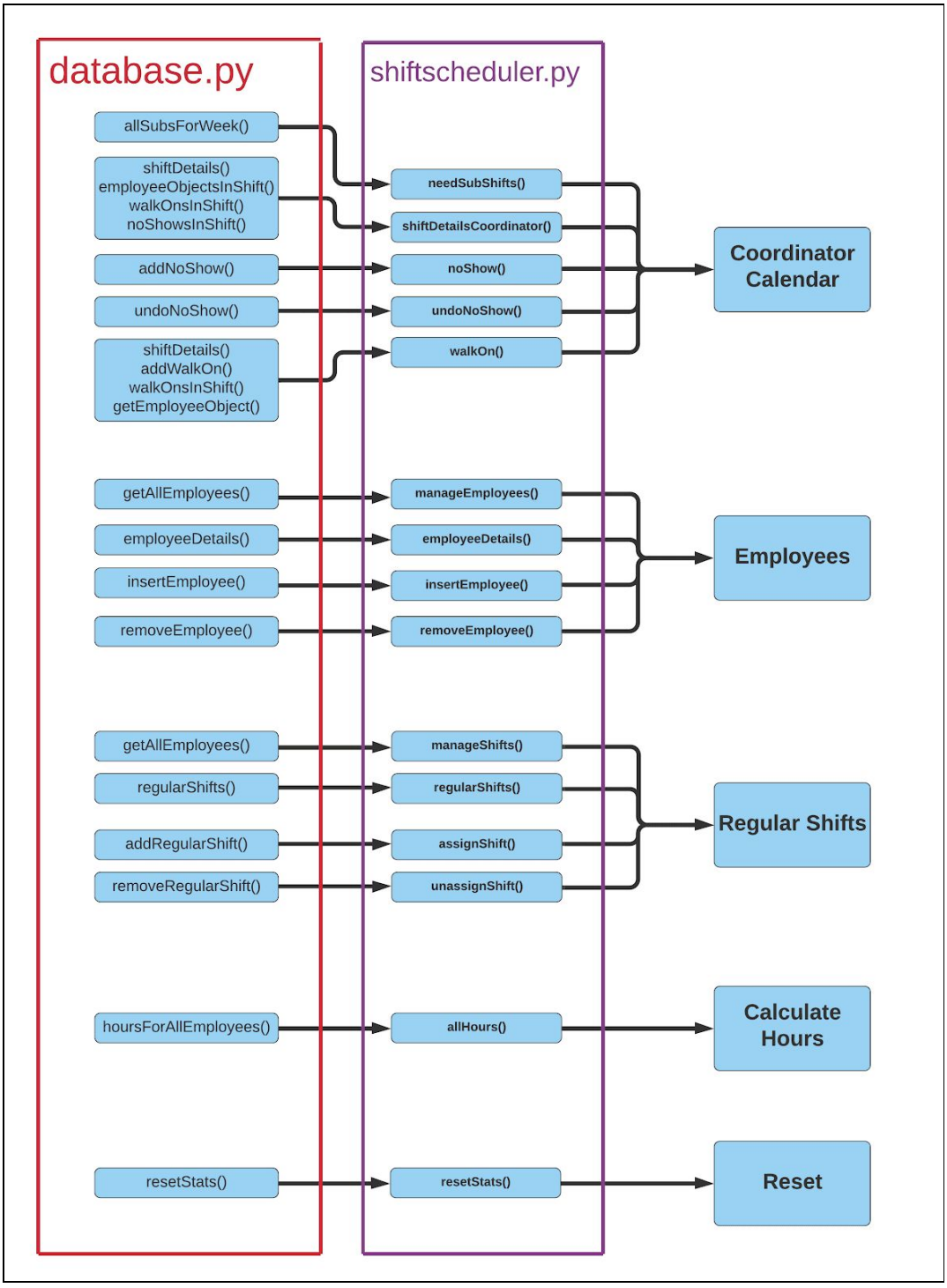


Figure 7: Communication between Coordinator frontend pages and backend

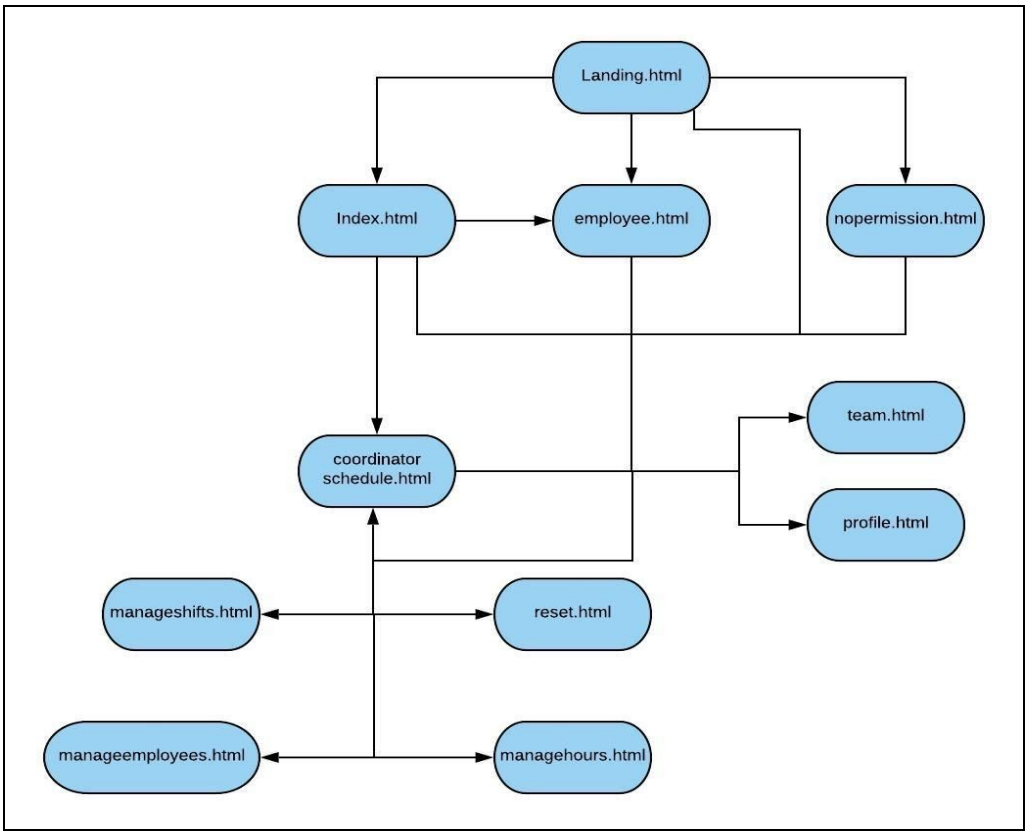
- **sendemail.py:** This file contains a function called mail_it() which uses the Mail and Message objects from the flask_mail module to send emails to all Employees when there is a sub request. The email is being sent from shiftscheduler@princeton.edu.

- **CASClient.py:** This file contains a class called CASClient which has methods authenticate and logout that are being used in the login() and logout() functions in shiftscheduler.py. Using this file, we were able to integrate our application with the Princeton CAS authentication system.

- **Communications:**
    a) **Database - database.py:** As mentioned earlier, connect() and disconnect() methods use the URL of the Heroku Postgres server to create a connection. Later, methods that query/modify the database create a cursor and execute SQL statements with this cursor. ShiftScheduler uses prepared SQL statements to protect itself from injection attacks.
    b) **database.py - employee.py:** database.py imports the Employee class from employee.py.
    c) **database.py - shift.py:** database.py imports the Shift class from shift.py.
    d) **database.py - shiftscheduler.py:** shiftscheduler.py imports the Database class from database.py.
    e) **shiftscheduler.py - sendemail.py:** shiftscheduler.py imports the mail_it function from sendemail.py.
    f) **shiftscheduler.py - CASClient.py:** shiftscheduler.py imports the CASClient class from CASClient.py.
    g) **shiftscheduler.py - Frontend:** The communication between shiftscheduler.py and the frontend is established through GET or POST requests. For pages that require communication with the backend while loading, Javascript's setup() function is used.. The results are loaded into the web page dynamically using AJAX. Also, JSON is used to transfer relevant data from the backend into the front end.

3) **Frontend:**

- **landing.html**
    - This is the HTML template for the landing page "/" or "/landing" of our application. You will be directed here if you go to https://shiftscheduler.herokuapp.com/.
    - On the page, there is a login button on this page that links to "/login". A GET request is sent to the backend once the button is clicked since the button is a link formatted like a Bootstrap button. There is also a footer that links to "/team".
- **index.html**
    - This is the HTML template for the index page "/index" of our application.
    - After the login, the user will be redirected to this page if they are listed as both a coordinator and an employee. If they are only an employee, they do not have access to this page. This is the "Home" page for coordinators. If the user clicks "Home" in the navbar dropdown or if the user clicks on the ShiftScheduler logo in the navbar, the user will be redirected to this page.
    - The index page consists of a header, footer and two buttons: Employee and Coordinator. Each will take the user to the view for the respective role. The employee button sends a GET request to "/employee" and the coordinator button sends a GET request to "/coordinatorschedule."



Possible connections between html files.

- **profile.html**
    - This is the HTML template for the profile page "/profile" that any user can access from the dropdown menu in the navbar.
    - The profile page is accessible by all employees. This page consists of a Bootstrap card with details (the user's netid, name, position, hours worked in the pay period and their regular shifts) and a "Back to Employee Calendar" button that will redirect to "/employee."
    - All details except for the regular shifts are passed into the template using Jinja2. The regular shifts information is pulled into the page dynamically using AJAX. In the setup() function which is called on ready of the document, a function called getShifts() is called in which we send a GET request for "/regularShifts" with argument netid. The responseTxt from this request is a JSON list of shifts in the form of "dayOfTheWeekNum-taskid," e.g. "0-1" would represent a Monday Dinner Manager shift. HTML formatting of this data is handled in JavaScript and for each shift, a list item added to the card. This is done only once on setup.
- **employee.html**
    - This is the HTML template for the employee calendar page "/employee."
    - It is the homepage for employees who are not coordinators. employee.html is the template for the calendar that all employees have access to. The main component in this page is the calendar which is implemented as an HTML table. Each meal has its own column and each section (dish/dinner/brunch) is a cell in this table. Each shift in each cell is formatted as a Bootstrap list-group-item and has a link associated with it. The other big HTML components in this page are the three Bootstrap modals that are hidden when the page first loads, but show appropriately as the user clicks on shifts.
    - The headers label each column with its day and meal, eg "Monday Dinner". Each of these column headers contains a class representing the day of the week, eg "mon" or "tue." Every cell also has such a class. Highlighting of the current day is done on setup through matching the day of the week of the current date to the class of the appropriate cells.
    - Note that though we believe we have resolved all of the problems, the dates that are displayed may cause bugs in different time zones since some of the functions take UTC as a reference and perform conversions accordingly. The application is meant to be used in EST, as that is where the Butler/Wilson dining hall is located. On setup(), the getDates() function in the file is called. In this function, we take in the date of Monday (in ISO format) in the current week from the URL and calculate the dates for the rest of the week, placing them into the appropriate cells. If this page is called with no "monday" URL arguments, the Monday will be calculated in this function using today's date.
    - There are two black arrows at the top right of the page, above the "Sunday Dinner" column. These navigation buttons have event handlers installed on setup. On "click," depending on whether it is the forward or backward arrow, the desired week's Monday date is calculated and a GET request to 'employee' with the

calculated ‘monday’ argument is created and the appropriate week is loaded.

- There are three modals on this page, all are initially hidden and at any moment at most one of them is displayed. All three show shift details. The only difference between these three modals is the button in the lower right corner. Shifts that are blue will pull up the #yourShift modal, and these will have a “Sub-Out” button. Shifts that are red will pull up the #needSubShift modal and these will have a “Sub-In” button. All are initialized to the #otherShift modal which has a “Close” button, so shifts that are neither blue or red would have these. All of these modals, when a shift is clicked, send the GET request ‘/shiftdetails’ to the back end with arguments task id and formatted date. The return of this request is a list of shift details, namely the date, meal, task, start time, end time, and list of employees working the shift if relevant, and this is loaded to the body of the modal.
- On setup, the function getColors() is called, and we make connections to the backend to color the shifts blue (representing shifts assigned to the current user in the current week) and red (representing shifts that require subs in the current week) accordingly. First, a GET request is sent to the backend to ‘/needSubShiftsEmployee’ with argument ‘mon’ representing the date in ISO format of the Monday of the week in question. This query returns a list of shifts that require subs for the specified week. We change the colors of these shifts to red by adding the class “list-group-item-danger” to the appropriate shift. Then, a GET request is also sent to ‘/myShifts’ again with the same ‘mon’ argument. This returns a list of shifts of the current user in the current week. We color these shifts blue by adding the class “list-group-item-info” to them. In both cases the relevant shifts’ data-target is also changed to the appropriate modal. This is done to make sure the right modal and actions are displayed for the user if the shift is clicked.
- Each list item is interactive and the user can click on it to open a modal with shift details. On setup, through Javascript, JQuery event listeners are installed on every list item representing shifts.
- An event listener is installed on setup() to the “Sub-In” and “Sub-Out” buttons. When the “Sub-In” button is clicked, a GET request is sent to the backend with the date and taskid of the currently pulled-up shift. The response will be a success message or a failed message. This will be loaded into the footer of the modal to replace the sub-in button and after a short pause the page reloads and the modal is hidden again. On reload, the shifts are again colored accordingly, ie if the sub-in request succeeded, that shift should now be colored blue.
- The sub-out request functions the same way, except the GET request is sent to ‘/subOut’ instead of ‘/subIn,’ and if successful, the shift would be colored red.
- Note that once any button is clicked and a GET request is made, the button is disabled until the request returns and is loaded into the footer to prevent errors that may arise from double clicking.

- **coordinatorschedule.html**
    - This is the HTML template for the “Calendar” tab in the coordinator view “/coordinatorschedule”.
    - The coordinator schedule functions very similarly to the employee schedule. The main difference is that there is only one type of modal on this page; however, there are more functionalities associated with that one type of modal. Another difference is that there are no blue-colored shifts and the only coloring is red, since the coordinator calendar is meant to be from a managerial perspective rather than of an individual employee looking at his/her shifts.
    - Similarly to the employee calendar, when a shift is clicked a modal is toggled. The information to populate the modal’s body is pulled from the backend, sending a GET request to ‘/shiftdetailsco’ (instead of ‘/shiftdetails’ as in the employee calendar) with the same arguments: the task id and formatted date of the requested shift.
    - Event listeners are installed to buttons of class noShow and undoNoShow using JQuery. When a noShow button is clicked it sends a GET request to ‘/noShow’ with the netid and shift id as arguments. If the request is successful the returned value will be html code for a red ‘no show’ button. This is loaded to the page to replace the ‘mark no show’ button.
    - If the red no show button is clicked to undo the no show, a GET request is sent to ‘undoNoShow’ to the backend similar to the noShow with arguments netid and shift id.
    - If the “Add a Walk-On” button is pressed when the input field is empty, an error is given. Otherwise, an AJAX request is sent to the back end to ‘/walkOn’ with arguments date, taskid and netid. If the request is successful, the back end returns the first and last name of the employee which is added to a list of walk ons dynamically in the modal. Otherwise, an error message is loaded into the footer.
    - It is not possible to add walk ons or mark an employee as a no show for the future since the shifts have not occurred yet. Thus, walk on and no show buttons for future dates (including the current date) are disabled. This is done in the function getModalData() that’s called on setup, which installs all the event handlers for when the modal is toggled.
    - One minor bug currently exists in this code: for each employee, an action (ie marking or undoing a no show) can only be done twice without dismissing this modal. This means one can only mark no show and undo it once or, undo a no show and mark a no show. As many walk-ons as desired can be added to a shift. This should not be a problem in practice, as we see no reasonable cases where you would need to mark and then undo a no-show more than once.
    - Once any button is clicked and a GET request is made, the button is disabled until the request returns and is loaded to prevent errors that may arise from

double clicking.
    - Navigation between weeks through the black arrows in the top right is implemented the same way as it is implemented in employee calendar.
- **manageemployees.html**
    - This is the HTML template for the “Employees” tab in the coordinator view “/manageemployees”. This page is used to view a list of all employees, to see their information, and to add and remove employees.
    - The leftmost column is a scrollable list of employees. Using Flask, when the template is rendered in ‘/manageemployees’ in shiftscheduler.py a list of employees is pulled from the database and passed into manageemployees as a list of Employee objects. Using Jinja2 templating, we print employee details for every employee. Each employee is inserted into a table in a single cell row with the attribute “link= ‘/employeeDetails?’” along with the appropriate argument netid and also of class employeeItem.
    - The middle column will display an employee’s details once one is clicked in the leftmost column. On setup(), an event handler is installed “on click” to all items of the class “employeeItem” where a GET request is sent to the backend to the value of the corresponding link attribute. The return value, formatted employee details, is then loaded into the middle column.
    - The last column is to add and remove employees. Once the “Add Employee” button is clicked, we robustly check the information through Javascript and JQuery. All information, i.e. the three text input fields and the radio button, must be filled. If there are any issues, red error messages will appear and the request will not execute. If all checks are passed, a GET request is made to ‘/insertEmployee’ with arguments employeenetid, firstname, lastname and manager. If the return status is successful a status message containing the netid of the employee and ‘was successfully added’ is printed under the Add Employee button. After a short wait, the page is reloaded. The added employee now appears in the list of employees. Note that currently there is no check if the netid actually exists in Princeton’s system so the return is successful unless there is something wrong with the database.
    - The remove employee section only has one text field which is for the netid. On setup(), removeEmployee() installs an event handler to the “Remove Employee” button. When the button is clicked, the function checks if the netid is an empty string and also doesn’t allow for the user to remove himself/herself. Red error messages will appear if appropriate. If all checks are passed, an ‘are you sure’ modal pops up when a removal request is made. If the user clicks ‘yes’, a ‘/removeEmployee’ GET request is sent to the backend with argument netid. If the user clicks ‘no’, the modal is dismissed, and no request is sent.
- **manageshifts.html**
    - This is the HTML template for the “Regular Shifts” tab in the coordinator view “/manageshifts”. manageshifts.html is formatted similarly to manageemployees.html with three columns. Only the leftmost is visible on start;

the other two populate once an employee's name is clicked in the leftmost column.
    - The left and middle columns are implemented the same way as in manageemployees.html. However, for the middle column here, when an employee's name is clicked, a GET request is made to '/employeeShiftDetails' with the argument netid. The returned HTML includes the shift details in addition to all of the employee's regular shifts and buttons to unassign them. These are implemented as links that send a GET request to '/unassign' with arguments day of the week and netid. If successful, the page reloads and displays the newly added details. Otherwise, the error message is loaded to display below the button.
    - When the "Add Regular Shift" button is clicked, the values of the dropdown menus are taken to send an AJAX request to '/assign' with arguments netid and shiftid. If the result that's returned from the AJAX request is successful, the page prints a success message and reloads with the new information after a short wait. Otherwise, the page prints the error message under the button.
- **managehours.html**
    - This is the HTML template for the "Calculate Hours" tab in the coordinator view "/managehours". It consists of the usual coordinator header and footer as well as two date form controls, text to explain the functionality, and a "Generate Report for Selected Period" button. When the button is clicked, the page handles it through Javascript and JQuery and displays a loading message right under the submit button. If the dates were empty or if the end date was earlier than the start date for the report, an error message is displayed. Otherwise an AJAX GET request is sent to '/allhours' with the arguments startDate and endDate. The return of this function consists of HTML code for a scrollable table that will display all the relevant information for the employees between the given dates.
- **reset.html**
    - This is the HTML template for the "Reset" tab in the coordinator view "/reset". If the button is clicked, a modal to make sure the user understands the request they are making is toggled through JQuery. If the user enters incorrect text, an error message appears under the input field and nothing is submitted. If the user enters the exact text into the input field and clicks the "Submit" button, an AJAX request to '/resetStatsLink' is made. The modal is dismissed and either a success message or a request failed message is displayed on the page under the button.
- **no permissions.html**
    - This is the HTML template for the no permissions page "/nopermissions." This page consists of the usual footer as well as a card explaining that the permissions are not valid for the page. It includes a button link that redirects to the landing page when clicked.
- **team.html**
    - This is the HTML template for the Team page. This page is accessible from the

footer of every page on the application. It includes a description for our project, a link back to the landing page, a card for every member of our team that includes a picture, name, link to contact (email application redirection) and a description. A Google form to report bugs is also linked in this page.

SECTION 3: INTERESTING DESIGN PROBLEMS

Backend:

One interesting design problem we encountered in the backend was related to the database design. Originally, we did not have the shift_assign table in our database. In order to get an employee's shifts, we were planning to get their regular shifts, add any sub-ins/walk-ons, and remove any sub-outs/no-shows. This approach was used for the alpha version. However, as we continued to add more functionalities to our application (such as calculating hours, undoing no-shows, etc.), we realized that this approach was not very efficient as it required many redundant database queries and it caused numerous corner-case bugs. Therefore, we added the shift_assign table to our database where we stored unique shift_id-netid pairings. Unfortunately for us, this warranted going through all of the backend methods and making them compatible with the new design. This was definitely a very interesting design problem; we initially thought keeping the number of database tables smaller would make the system simpler, but in practice it actually made it very convoluted. In conclusion, carefully thinking about database design and finalizing it early in the process is very important.

Another interesting design decision we made in the backend was related to calculating hours. Every time an employee clicks on their Profile page, we are recalculating their current hours in the pay period and updating the "hours" column of their row in the database. Similarly, when a coordinator selects a period and clicks on "Generate Report for Selected Period" on their "Calculate Hours" tab, we are calculating hours for all employees in that given period and updating their "hours" column in the database. In other words, the "hours" column of the employees table does not always represent the most recent worked hours in the current pay period, instead it contains the most recently "calculated" hours. But since Profile and Calculate Hours pages are the only places where the users can see hours and since we are recalculating the hours before accessing both of these pages, this design approach does not cause any problems. In hindsight, we did not really have to keep a "hours" column in the employees table since we are always recalculating hours before displaying them. However, since we did not observe any apparent efficiency problems, we decided not to change this approach that was already working. Later if we were to see efficiency problems with more users, we could modify the "hours" column to always contain the current number of hours worked in the most recent pay period (for having fast access to this important information), and recalculate hours (but don't update it on the employees table) when the coordinator inputs a different time period on their "Calculate Hours" tab.

Frontend:

For the design of the frontend, the most challenging parts to design were the employee and coordinator calendars. Initially, we had a classical calendar view in mind, i.e. one column for every day of the week, and our alpha reflected that. Also, in our alpha, we did not include dish shifts, so the weekdays each had three shifts and the weekends six. We planned on adding dish shifts by cutting the shift boxes in half and putting two types of shifts in the same box. However, once we tried to implement that we realized that it looked too cluttered and wasn't optimally minimalist. Thanks to the input of the Masi (the coordinator consulted throughout the design process) we decided to make each meal a column rather than each day, resulting in our current nine-column set-up, and abandoned the calendar view organized by hours. We think that this was definitely the right decision to make, as it's not too confusing and definitely makes the calendar less cluttered.

In addition, we wanted some type of pop-ups to display the information about select shifts. Our initial idea was to make some sort of dialog box appear when a shift is hovered over. However we realized that would not be very efficient on the user's side and probably hide other information that the user might want to look at. We also considered adding an additional column in which to display specific shift information, but ended up not doing that because it would add to the clutter. In the end, we decided to use Bootstrap's modals, which pop-up in the center of the page and darken everything else. Aesthetically, the modals match what we wanted pretty well; however, they were hard to work with when coding. For example, in the employee calendar, we decided to use three different modals, one for each type of shift. This meant we had to update the information dynamically, which posed a lot of problems. We were able to resolve them or find workarounds in the end, but it was very time-consuming. Also, in the coordinator view, we added more functionalities to the modal, such as dynamically loading in mark no-show and undo no-show buttons, which made things even more difficult.

One other problem we ran into was dates. We decided to use store dates in our database in ISO format and use date-time objects in Javascript and Python. However, we didn't realize that date-time objects in the two platforms are referenced a little differently. For example, Sunday is day 0 in JavaScript. JavaScript also uses UTC as a base time zone for most of its functions when really the local timezone is the relevant one, which was frustrating because we would get different errors based on the time of day where we were. Especially since we have a large time difference between our team members, this made tracking down errors pretty difficult since they would be hard to reproduce and would change depending on our meeting time. We believe that we have fixed all of the discrepancies, and in practice, it probably shouldn't be an issue because the application will only be used in the EST timezone.