# Scheduler\_App Codebase Analysis and Implementation Plan

## Codebase Overview

**Project Structure:** The **Scheduler\_App** is a Flask-based web application for hospital employee scheduling. The code is divided into a Python backend (app.py) and HTML/JS frontend templates under a templates/ directory. Key components include:

* **Flask App (app.py):** Defines data models (Employee, Schedule, TimeOffRequest) and API endpoints for managing employees, generating schedules, and handling time-off requests. It uses an SQLite database via SQLAlchemy[[1]](file://file_000000009b7461f79961f11bd0037eef#:~:text=class%20Employee%28db.Model%29%3A%20,in%20the%20scheduling%20system)[[2]](file://file_000000009b7461f79961f11bd0037eef#:~:text=class%20TimeOffRequest%28db.Model%29%3A%20,request%20submitted%20by%20an%20employee). On startup, the app initializes the database schema and runs a development server on port 5005[[3]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20__name__%20%3D%3D%20%27__main__%27%3A%20,run%28debug%3DTrue%2C%20port%3D5005).
* **Data Models:**
* Employee: Fields for name, email, and flags like is\_lead and nights\_only (instead of a single shift-type field) plus scheduling constraints like max\_hours\_per\_week and cannot\_work\_days[[4]](file://file_000000009b7461f79961f11bd0037eef#:~:text=Maximum%20number%20of%20hours%20the,datetime%20Timestamp%20of%20creation)[[5]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27employees%27%20id%20%3D,Boolean%2C%20default%3DTrue). (Note: Fields for things like **special schedule**, **shift preference**, **max consecutive days**, etc., are referenced in the UI but not present in this model.)
* Schedule: Represents an assigned shift for an employee (date, start/end times, type Day/Night, role label, overtime flag)[[6]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27schedules%27%20id%20%3D,Lead%2C%20D1%2C%20N1)[[7]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,schedule_date%3Dcurrent_date%2C%20shift_start%3Dshift_start%2C%20shift_end%3Dshift_end%2C%20shift_type%3D%27DAY%27%2C%20role%3Drole_label).
* TimeOffRequest: Stores employee PTO (paid time off) requests with start/end dates, shift type (Day/Night/Both), status, etc.[[2]](file://file_000000009b7461f79961f11bd0037eef#:~:text=class%20TimeOffRequest%28db.Model%29%3A%20,request%20submitted%20by%20an%20employee)[[8]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27timeoff_requests%27%20id%20%3D,Text%2C%20nullable%3DTrue).
* **Flask API Endpoints:**
* **Employee Management:** GET /api/employees returns active employees; POST /api/employees creates a new employee; PUT /api/employees/<id> updates an employee; DELETE /api/employees/<id> deactivates (soft-deletes) an employee[[9]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%27%2C%20methods%3D%5B%27GET%27%5D%29%20def%20get_employees%28%29%20,list%20of%20all%20active%20employees)[[10]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,try%3A%20if%20%27name%27%20in%20data).
* **Schedule Retrieval:** GET /api/schedule returns scheduled shifts in a date range[[11]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Fschedule%27%2C%20methods%3D%5B%27GET%27%5D%29%20def%20get_schedule%28%29%20,for%20a%20given%20date%20range)[[12]](file://file_000000009b7461f79961f11bd0037eef#:~:text=schedules%20%3D%20Schedule.query.filter%28%20Schedule.schedule_date%20,sch.to_dict%28%29%20for%20sch%20in%20schedules).
* **Schedule Generation:** POST /api/schedule/generate triggers generation of a new schedule for a given start date and number of weeks[[13]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Fschedule%2Fgenerate%27%2C%20methods%3D%5B%27POST%27%5D%29%20def%20generate_schedule_endpoint%28%29%20,for%20a%20given%20date%20range)[[14]](file://file_000000009b7461f79961f11bd0037eef#:~:text=Schedule.query.filter%28Schedule.schedule_date%20%3E%3D%20start_dt%2C%20Schedule.schedule_date%20,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments).
* **Time Off Requests:** GET/POST /api/timeoff list all requests or create a new request; PUT /api/timeoff/<id>/approve and /deny to approve/deny a request[[15]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%27%2C%20methods%3D,upper)[[16]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%2F,400%20req.status%20%3D%20%27APPROVED).
* *(Shift trade endpoints are notably missing in the current code – more on this below.)*
* **Scheduling Algorithm:** The core scheduling logic is implemented in generate\_schedule(start\_date, days) inside app.py. This function uses a **greedy assignment** strategy to fill each day’s Day and Night shifts while enforcing business rules[[17]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,contiguous%20range%20of%20days)[[18]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,when%20necessary%20to%20meet%20coverage). It handles lead and night-only employees, weekly hour caps, required rest (10 hours) between shifts, max 5 consecutive work days, and minimum coverage (4 on day shifts, 3 on nights)[[19]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shifts%20while%20enforcing%20the%20following,rules)[[20]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,when%20necessary%20to%20meet%20coverage). The generated Schedule objects are then saved to the DB by the endpoint[[21]](file://file_000000009b7461f79961f11bd0037eef#:~:text=db.session.commit%28%29%20,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments).
* **Frontend Templates:** HTML files (e.g. index.html, dashboard.html, employees.html, schedule.html, timeoff.html, shift\_trades.html, rules.html) use a common base layout (base.html) and include embedded JavaScript to call the API. The UI provides pages for a dashboard, managing employees, viewing schedules, handling time-off requests, and initiating shift trades. Key frontend scripts include:
* **Dashboard (dashboard.html):** Loads the week’s schedule and pending PTO requests via API and displays stats and charts[[22][23]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK). Also allows generating a new schedule (opening a modal to choose start date and weeks)[[24][25]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK).
* **Employees (employees.html):** Lists all employees in a table and includes forms to add a new employee or edit an existing one[[26][27]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). JS functions call GET /api/employees to populate the list[[28]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) and (intended to) call POST /api/employees to add new staff[[29]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). Edit and delete actions are present in the UI but not fully wired to the backend (more on this below).
* **Time Off (timeoff.html):** Provides a form to submit a PTO request and a table of all requests with approve/deny buttons for pending entries[[30][31]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH). It calls GET /api/employees to populate the employee dropdown and GET /api/timeoff to list requests[[32][33]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH).
* **Shift Trades (shift\_trades.html):** UI to request a shift swap between two employees. This page is currently using **mock data** in its JS (the code comments indicate it’s not yet connected to any API)[[34][35]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj). It shows a form to choose two employees and their shifts to trade, and a table of trade requests with approve/deny options, but no real backend logic is in place.

Overall, the codebase provides a good starting structure aligned with the **project’s requirements** (hospital PBX scheduling system). However, in its current state, several functions “don’t work” as expected due to bugs, unimplemented features, or mismatches between frontend and backend. Below, we detail these issues and propose solutions, including how to leverage the AI scheduling logic to handle PTO (“PTO reshuffling”) and other requirements.

## Identified Issues in Current Implementation

Despite having a substantial code framework, **many functions are not working properly**. The primary reasons are:

### 1. Flask Route and Endpoint Bugs

* **Duplicate Function Names:** The Flask routes for the home page (/) and the dashboard (/dashboard) are defined using the *same* function name root\_page. This likely causes a conflict or overwrites one of the routes. In app.py we see both routes using def root\_page():[[36]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app). Flask endpoints default to the function name, so this duplication means the second definition might override the first or throw an error (view function mapping conflict). This can prevent the app from starting or the / route from working. The intention was probably to name the second function differently (e.g. def dashboard() for the /dashboard route).
* **Inconsistent Endpoint Names in Templates:** The navigation bar in base.html uses Jinja checks like {% if request.endpoint == 'dashboard' %} to mark the active page[[37]](https://drive.google.com/file/d/1rY4_AhLweywarRv1dUOTooV21tfe1MVp). However, because the dashboard view function is not actually named "dashboard" (it’s using root\_page erroneously), the condition fails. Similarly, the schedule page link expects an endpoint 'schedule\_view'[[38]](https://drive.google.com/file/d/1rY4_AhLweywarRv1dUOTooV21tfe1MVp), but the actual function is named schedule[[39]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app). These mismatches mean the active menu highlighting won’t work and indicate minor naming inconsistencies. (This is mostly cosmetic, but fixing the endpoint names will resolve it.)

**Plan to Fix:** Rename the /dashboard route function to a unique name (e.g. dashboard()), and ensure each page route has a distinct function name. Update the base.html conditions to match the new endpoint names (e.g., use if request.endpoint == 'dashboard' if the function is named dashboard). This will remove any routing conflicts and correct the nav highlighting.

### 2. Frontend–Backend Data Mismatch

Several functions “don’t work” because the data exchanged between the UI and API is not aligned:

* **Employee Creation Fields:** When adding a new employee via the UI form, the JavaScript collects fields including name, email, shift\_type (Day/Night/Both from a dropdown), hours\_per\_week, special\_schedule (e.g. Lead, Legal Cap, New Hire), max\_consecutive\_days, min\_rest\_hours, and cannot\_work\_days (checkboxes)[[40]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). It then sends this JSON to POST /api/employees. **However, the backend expects a different schema:** it looks for name, email, and uses boolean flags is\_lead and nights\_only instead of a single shift\_type, and max\_hours\_per\_week instead of hours\_per\_week[[41]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee%20%3D%20Employee%28%20name%3Ddata,%29%20db.session.add%28employee)[[42]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20%27nights_only%27%20in%20data%3A%20employee,to_dict%28%29%29%20except%20Exception%20as%20e). The current add\_employee implementation in app.py does **not handle** fields like shift\_type or special\_schedule at all. As a result:
* Selecting "Night Only" or "Day Only" in the UI doesn’t set the corresponding nights\_only flag on the backend (and there is no field for "day-only", meaning that preference is lost entirely).
* Marking an employee as "Lead/Supervisor" in the form’s special schedule does not set is\_lead=True – the JSON is sent with "special\_schedule": "LEAD", but the backend simply ignores unknown fields. Thus, the user cannot actually designate a lead operator through the UI (Patty would not be marked as lead in the DB).
* The hours\_per\_week value from the form is never read by the backend. The code uses max\_hours\_per\_week = data.get('max\_hours\_per\_week', 40)[[43]](file://file_000000009b7461f79961f11bd0037eef#:~:text=is_lead%3Dbool%28data,get%28%27cannot_work_days%27%2C), so it defaults to 40 every time since max\_hours\_per\_week isn’t in the JSON (the JSON uses hours\_per\_week). This means any custom hour limit (e.g. for Vicki’s 20-hour cap) isn’t saved – Vicki would incorrectly default to 40 hours unless manually edited in the database.
* **API Response Format:** The frontend code expects certain responses to contain a success flag, but the backend doesn’t provide it consistently:
* After adding an employee, the JS does if (response.success) { ... } else { showAlert("Failed to add employee") }[[44]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). But the POST /api/employees endpoint returns the new employee’s data directly (and a 201 status) with no {"success": true} wrapper[[45]](file://file_000000009b7461f79961f11bd0037eef#:~:text=db,error%27%3A%20str%28e%29%7D%29%2C%20400). Consequently, response.success is undefined and the UI always goes to the “Failed to add” path – the user sees an error even though the employee was actually created in the database. The employee list then doesn’t refresh (because the code only calls loadEmployees() on success), making it appear like nothing happened.
* The time-off request submission has a similar pattern: if (response.success) ... else show error[[46]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH). The POST /api/timeoff endpoint also returns the created request object (with status 201) but no success flag[[47]](file://file_000000009b7461f79961f11bd0037eef#:~:text=reason%3Dreason%2C%20%29%20db,to_dict%28%29%29%2C%20201). Thus, every new PTO request shows a failure message on the UI, even though the request likely saved. The list of requests won’t refresh in the UI in this case either.
* By contrast, some other API responses do include a success field (e.g., schedule generation returns {"success": true, "generated\_shifts": N}[[48]](file://file_000000009b7461f79961f11bd0037eef#:~:text=new_assignments%20%3D%20generate_schedule%28start_dt%2C%20total_days%29%20db,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments), and approving/denying time off also returns {"success": true, ...}[[16]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%2F,400%20req.status%20%3D%20%27APPROVED)[[49]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH)). The inconsistency can confuse the client-side logic.
* **Employee Update/Delete:** The UI has buttons for editing and removing employees, but these functions are stubbed out:
* The **Edit form** is shown and allows changing fields, but on submit it currently does *not* call the API. In employees.html, the edit form submission handler simply displays a success alert and reloads the list without actually sending a PUT request[[50]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). So, edits do not persist. (The backend PUT /api/employees/<id> exists to update fields[[51]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,name)[[52]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee.nights_only%20%3D%20bool%28data,to_dict%28%29%29%20except%20Exception%20as%20e), but it’s never invoked.)
* The **Remove** button click calls removeEmployee(id), which also just pops a confirmation and then immediately shows a "removed successfully" alert and reloads the list[[53]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). It doesn’t call the DELETE /api/employees/<id> endpoint at all. Thus, employees are never actually deactivated in the database. (The backend deactivate\_employee sets active=False for the employee[[54]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,404%20employee.active%20%3D%20False), but again, it’s not used by the UI.) After a page refresh, the "removed" employee would still appear since the DB wasn’t updated.

In summary, **the frontend and backend are out of sync on critical operations**, causing features to appear broken. New employees or PTO requests seem to fail (due to response handling), and editing/removing employees does nothing (not wired up). Similarly, special attributes like lead or night-only aren’t applied because the backend doesn’t interpret the form data correctly.

**Plan to Fix Data Mismatches:** We need to align the API with the UI (or vice versa):

* **Unify Field Names:** Update the add\_employee endpoint to handle the fields the UI sends. For example:
* Accept shift\_type from the JSON. Map it to the boolean flags: e.g. if shift\_type == "NIGHT" then set nights\_only=True; if shift\_type == "DAY" then we might introduce a new flag for day-only (or handle as nights\_only=False but also ensure the person is not scheduled at night — more on this in the algorithm section). If shift\_type == "BOTH" (the default), then nights\_only=False (and not lead unless specified separately).
* If special\_schedule == "LEAD", set is\_lead=True for that employee. The backend currently ignores special\_schedule, but we can easily add logic to handle known codes (LEAD, LEGAL\_CAP, NEW\_HIRE). For MVP, setting is\_lead is crucial so that the designated lead (e.g., Patty) is actually treated as such by the schedule generator. Other special codes can be stored (perhaps add a special\_schedule column to Employee) for future use or simply acknowledged without effect if we don’t implement special logic for them yet.
* Use hours\_per\_week from the request to set max\_hours\_per\_week. We can change the backend to read data.get('hours\_per\_week') if present (or simply send the JSON key as max\_hours\_per\_week from the UI – but since the UI code is already written, adjusting the backend is fine). This will allow setting custom weekly hour limits (e.g., 20 for part-timers).
* Add fields for max\_consecutive\_days and min\_rest\_hours to the Employee model or at least store them. Currently the model has these values effectively hard-coded (5 consecutive days, 10 hours rest), but since the UI collects them per employee, it implies flexibility. We can extend the Employee table (add max\_consecutive\_days and min\_rest\_hours columns) and save these values. Initially, we might still enforce uniform rules (like everyone needs 10h rest, 5 days max) to satisfy the original requirements, but storing these per employee allows future customization. If adding new columns is not feasible right now, as a temporary measure, we could ignore these fields (or validate they are 5 and 10 as expected defaults). However, given we are **“close to productionalizing”**, it’s better to incorporate these properly for completeness.
* Handle cannot\_work\_days properly. The UI gathers an array of days (Mon, Tue, etc.) and currently JSON-encodes it as a string before sending[[55]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). The backend expects a list for cannot\_work\_days and then json.dumps it again[[56]](file://file_000000009b7461f79961f11bd0037eef#:~:text=nights_only%3Dbool%28data,). This double encoding could lead to a stored value like "[\"Mon\",\"Fri\"]" (string within a string). Instead, the UI could send the array as-is (not stringified), and the backend should json.dumps that into a string to store. Or simply send as an array and modify backend to store appropriately. We should test and ensure that an employee’s cannot\_work\_days is stored as a JSON text (like ["Fri"]) in the DB and correctly parsed when used in the schedule algorithm (the generator does json.loads(e.cannot\_work\_days or '[]') each day[[57]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,if%20weekday%20in%20cant_work%3A%20continue), so it expects a JSON string in the DB, which is correct as long as we only encode once).
* **Consistent API Responses:** Standardize the API responses for create/update operations:
* For **create** actions (adding employee, submitting time-off), modify the Flask endpoints to return a success flag and/or message. For instance, after creating an employee, return jsonify({"success": True, "employee": employee.to\_dict()}). The UI code is already looking for response.success. By adding this, the UI will recognize success and proceed to close the modal and refresh the list[[58]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). Similarly, for new time-off requests, return {"success": True, "request": req.to\_dict()}. This way, the front-end will show the success alert and refresh the data[[59]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH). (Alternatively, we could change the front-end to not require a success field and treat any 2xx HTTP response as success. But adding the flag on the backend is straightforward and makes the API responses uniform with the ones already using {"success": ...}.)
* For **update/delete** actions, also return a JSON with success:true (and perhaps the updated object or a status message). The UI stubs for edit/delete can then be replaced to call these endpoints and handle the response in the same way.
* **Wire Up Edit and Delete:** Implement the JavaScript calls for editing and deleting employees:
* On edit form submit, gather the changed fields and send an API PUT /api/employees/<id> request with those fields. Include at least name, email, shift\_type (or the equivalent flags), hours/week, etc., that were possibly changed. Upon success (200), refresh the employee list and show a success message. The backend update\_employee already applies only provided fields[[60]](file://file_000000009b7461f79961f11bd0037eef#:~:text=data%20%3D%20request,is_lead%27%5D%29%20if%20%27nights_only%27%20in%20data)[[52]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee.nights_only%20%3D%20bool%28data,to_dict%28%29%29%20except%20Exception%20as%20e), so we can send only the fields that changed or just send all fields from the form (the backend will ignore any it doesn’t use). Notably, update\_employee can handle changes to is\_lead, nights\_only, etc., but since the UI form uses the combined fields (shift\_type, special\_schedule), we might need to translate those before sending (similar to the add flow).
* On delete (remove) click, call API DELETE /api/employees/<id>. If that returns success, remove the row from the table or reload the list. The backend will mark the employee inactive[[61]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,status%27%3A%20%27success). We should then ensure our GET /api/employees continues to return only active ones (it does by filter active=True[[62]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employees%3A%20List,emp.to_dict%28%29%20for%20emp%20in%20employees)). After deletion, the UI list will exclude that employee on refresh. (It might also be wise to provide some visual indication in the UI if needed for inactive employees, but since we’re filtering them out entirely, it’s okay.)

By fixing these data handling issues, the basic **Employee management and PTO request flows will function correctly**: adding a new employee will immediately reflect in the UI (no false error), editing will persist changes (e.g., marking someone as lead), and removing will truly deactivate the person. Similarly, submitting time-off will show up in the pending list without manual refresh, etc. This addresses a large part of the “nothing works” perception, as currently these forms appear broken.

### 3. Incomplete Feature Implementations

Some features are only partially implemented, leading to non-functional UI components:

* **Shift Trades (Swap Requests):** The shift trade functionality is listed in the requirements and a UI exists, but it is completely using placeholder logic right now:
* The trade request form doesn’t load actual shift options. The <select id="original-shift"> and <select id="target-shift"> are never populated with the user's real shifts. The code that should load available shifts for the selected employees is not written (likely intended but left out). Instead, the page’s script calls loadShiftTrades() which currently just inserts some mock trade requests into an array[[34]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) and displays them. There is no API call to fetch real trade requests from the server, and indeed no backend model or endpoint for trades is defined yet.
* Approving or denying a trade in the UI does not contact the server; it just updates the mock data array and refreshes the table locally[[63][64]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj). Similarly, submitting a new trade request pushes a new object into the shiftTrades array with a made-up ID and status 'PENDING'[[65]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj), but does not save anything to a database or notify anyone.

Essentially, the **shift trade system is a stub** – it’s not wired to a backend at all. This means none of the trade-related functionality actually persists or affects the real schedule, which would definitely be noticed as “not working” if someone tries to use it.

* **Schedule “Rules” Page:** There is a rules.html (likely meant to display scheduling rules or allow configuring them). Depending on its content (not fully reviewed here), it might also be static or incomplete. Since it’s not critical to core operation, we can assume it’s informational. (But ensure the /rules route works – it is defined to render rules.html[[66]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app), so as long as the template exists, it should display.)
* **Email Notifications and Exports:** The requirements mention email notifications and exporting schedules, but no code exists for these (which is fine for MVP). We simply note that these features are not implemented, but they wouldn’t cause UI errors – they’re just not present yet.

**Plan to Complete Features:**

* **Implement Shift Trade Backend:** To make shift trades functional, we need to introduce a new model and API:
* Create a ShiftTrade model corresponding to the shift\_trades table described in the requirements[[67][68]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). At minimum, store: requesting\_employee\_id, target\_employee\_id, original\_schedule\_id, trade\_schedule\_id, reason, status, timestamps. (The UI currently is not using schedule IDs, but rather textual shift descriptions; we should change the UI to actually reference specific shift entries from the schedule for accuracy.)
* Add API endpoints: e.g., GET /api/trades to list trade requests (possibly filter by status or by requesting/target employee), POST /api/trades to create a new trade request, and PUT /api/trades/<id>/approve (or a general update endpoint) to approve/deny trades. These should mirror the pattern used for time-off requests.
* When a trade is approved, the system should **swap the shifts between the two employees**. This is the hardest part: we must update the Schedule entries. We can locate the two schedule records by their IDs (original\_schedule\_id and trade\_schedule\_id in the ShiftTrade). Then perform the swap: assign the requesting employee to the target’s shift and vice versa. In practice, swapping could be done by exchanging the employee\_id fields of those two schedule records (if we ensure they cover the same date or otherwise, but since a trade could be across different days or shifts, it’s essentially each takes over the other’s shift). We must also consider updating the role or shift labels if those are tied to specific people (probably not, roles like D1, N2 can remain the same, just who fills them changes).
* After swapping, mark the trade request status as 'APPROVED' and maybe store an approved\_at. If a trade is denied, we simply mark it denied and no changes to schedule occur.
* **Adjust Frontend for Trades:** Remove the mock data. Instead:
  + Populate the "Your Name" and "Trade With" <select> options using real employees (the UI already calls loadEmployees() and populates those selects[[69][70]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj), so that part is okay).
  + Populate "Shift to Trade" options once the user selects their name. This likely requires an API call to get that employee’s upcoming scheduled shifts. We could create an endpoint like /api/schedule?employee\_id=X&start\_date=...&days=... or simply fetch a week or two of schedule (the UI might just call the existing GET /api/schedule and filter it client-side). For simplicity, we might fetch the full current schedule (or the next few weeks) and then filter in JS for shifts belonging to the requesting employee, populating the dropdown with something like "2025-10-05 Day (D1)" as value/label. Do the same for the target employee's available shifts.
  + On form submit, call POST /api/trades with the selected requesting employee, target employee, and the two schedule IDs (plus reason). The backend will create the trade entry and return success.
  + Refresh the trade list (GET /api/trades) to include the new request in the table.
  + For approving/denying, call the PUT endpoints when the manager clicks the buttons, rather than just local updates. On success, maybe refresh the schedule view or at least the trades list.
  + **Integration with schedule**: once a trade is approved and the schedule table updated, those changes should reflect in the schedule page and elsewhere. We might want to trigger a refresh of the schedule data on the dashboard or schedule page after a trade is approved (to show the new assignments). This could be done via events or simply manual refresh by the user for now.

Implementing shift trades fully is a significant addition, but it will make that part of the app functional. It’s acceptable to start with minimal functionality: e.g., assume trades are always within the same week or between similar shift types to avoid complex validation. The requirement’s table suggests storing original and trade schedule IDs, which is a robust approach.

* **Verify/Complete Rules Page:** If the rules.html is meant to display configured rules (like current constraints: min staff, hours, etc.), we should ensure it’s populated either from constants or from the database if such settings exist. It might also be a static informational page listing the rules (which could be directly taken from the requirements). If static, no action needed beyond confirming it loads. If dynamic, we’d need to supply the data (perhaps via a small API or context on render). Given time, likely this page is just a static reference of business rules.
* **Testing UI After Fixes:** Once the above fixes are in place, thoroughly test each workflow:
* Add a new employee (with various combinations of shift prefs and special schedules) and see that they appear in the list with correct attributes (check the “Shift Type” and “Status” columns in the table). Ensure a lead indicator is stored (maybe add a visual badge for leads if desired).
* Edit an existing employee (change their hours/week, toggle lead or night-only via the form) and verify the changes stick (perhaps by checking the database or seeing their new status in the list).
* Remove an employee and confirm they disappear from the list (and the DB active flag flips to 0).
* Submit a time-off request and confirm it appears in the table as PENDING and also reflects on the calendar (the calendar highlights approved time off in red; pending requests won’t show on the calendar until approved, by design[[71][72]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH)).
* Approve a time-off request and ensure it moves to approved status in the table (the UI will call the API and refresh the list[[49][73]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH)).
* Generate a schedule and confirm that the shifts populate on the dashboard (the chart and weekly grid should update).
* Try the shift trade flow: create a trade request and then approve it. Then check the schedule to see if the shifts swapped between employees for those dates.

### 4. Time-Off (PTO) Not Integrated into Scheduling

Perhaps the most important logical gap is that **approved time-off requests are not accounted for in the schedule generation or updates**. According to the requirements, the system should “automate ... time-off management” and the AI engine’s logic flow starts with **“1. Process approved time-off requests”**[[74]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). In the current implementation:

* The generate\_schedule() function does **not** check any time-off records. It only considers the employees’ static restrictions (like cannot work certain weekdays)[[75]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,last_shift_end%27%5D%3A%20days_since_last%20%3D%20%28current_date). So if an employee has approved PTO for a given date, the algorithm as written could still assign them a shift on that date – a major conflict.
* When a time-off request is approved via the API, the code simply marks the request status and timestamp[[16]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%2F,400%20req.status%20%3D%20%27APPROVED) but does nothing to adjust any existing schedule that might be affected by that PTO. For example, if John Doe was originally scheduled on 2025-10-10 day shift and then a PTO request for John on 10/10 is approved, the system currently **leaves John on the schedule** for that day. There is no automatic reshuffle or notification to fill that gap.

This disconnect is likely why the user expects “AI to run scheduling and PTO reshuffling” – i.e. when PTO is approved, the system should intelligently reassign shifts or regenerate the schedule to accommodate the absence.

**Plan to Integrate PTO in Scheduling:**

* **Prevent Assigning Employees on Their PTO:** Update the schedule generation algorithm to skip employees who have time off during the target period. We can query all **approved** time-off requests once at the start of generation for the range being scheduled. For each employee, determine the dates (and shift types) they will be out. Then, in generate\_schedule loop:
* For each day, before assigning shifts, remove from the candidate pools any employee who has PTO covering that date **and that shift type**. For example, if an employee is on PTO for 10/10 and requested “DAY” only, we should not assign them a day shift on 10/10 (but they could still take a night shift if they’re otherwise eligible and willing – this nuance might be ignored for simplicity by treating any PTO as full-day off, or we can implement the shift-type check since the data supports it).
* This “PTO block” needs to be applied both for day\_pool and night\_pool construction. Concretely, we can extend the existing check for cannot\_work\_days[[57]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,if%20weekday%20in%20cant_work%3A%20continue) to also check a new set like pto\_days[employee\_id] that contains dates they are off. If current\_date is in that employee’s PTO set (and if shift type matches or is BOTH), skip them.
* By doing this, the generated schedule will never assign someone on a day they have approved PTO, which satisfies the primary requirement that PTO is respected.
* **Automatic Reshuffling on PTO Approval:** Handling PTO in generation is great for *future schedules*, but what about a PTO approved after a schedule is already generated/published? We have two approaches:
* **Manual Regeneration:** Simplest approach is to prompt the user (scheduler manager) to regenerate the schedule for the affected period whenever a PTO is approved. However, this is not very “AI-driven” and can disrupt the entire schedule for that week.
* **Targeted Adjustment:** Ideally, upon approving a time-off request, the system should **remove** the employee from any shifts that fall in their PTO range and try to fill those open shifts with someone else (while obeying rules). This is essentially a **micro scheduling problem** for the specific dates in question. We can implement a function (invoked after approval) that:
  + Finds all Schedule entries overlapping the PTO (e.g., for each date from start\_date to end\_date of the request, find any shift in the Schedule table where employee\_id is the one on PTO and schedule\_date is that date and (if shift\_type is specified in PTO, match that too unless PTO was BOTH)).
  + Delete or mark those schedule entries as needing coverage. (We might remove them entirely, or set is\_coverage=False or employee\_id=NULL if the schema allowed, but simpler is to delete them and later re-generate just those slots.)
  + Invoke the scheduling algorithm to fill the specific vacant days/shifts. We could call a modified generate\_schedule for each affected day or for the span of the PTO, with that employee temporarily marked inactive or unavailable. For example, if PTO is for 3 days, call generate for those 3 days only (starting that day) to get replacement assignments. We must be careful not to mess up other assignments on those days – perhaps we should only generate for the exact number of required people minus one (because one person was removed) and ensure others already assigned that day remain intact. This is complex with the current algorithm because it generates a full set from scratch for a day.
  + A simpler heuristic: find a replacement from the pool for each dropped shift. We could iterate each dropped shift and try to assign another employee to it who is available (not already working that day, under hours limit, etc.). This can be done by checking the state that was used for generation or by just applying rules at that moment. While not as optimal as re-running the whole day’s scheduling, it localizes the change.

Given the timeframe, an easier interim solution is: **notify the user that a schedule conflict exists and recommend regenerating**. But since we want an automated solution:

**Proposed Implementation (Automated):** When a PTO is approved: - Immediately call the schedule generator for the range of that PTO (e.g., if PTO is for a week, regenerate that week). This is brute-force but ensures the new schedule respects the PTO. The endpoint could do this behind the scenes: after req.status = 'APPROVED' and committing, call generate\_schedule(start\_date= PTO.start\_date, weeks= n) where *n* covers the PTO range. This will delete and rebuild the schedule for that span[[76]](file://file_000000009b7461f79961f11bd0037eef#:~:text=end_dt%20%3D%20start_dt%20%2B%20timedelta,commit). **Important:** The current generator code, when called via /api/schedule/generate, **wipes out** any existing schedule entries in the target range[[76]](file://file_000000009b7461f79961f11bd0037eef#:~:text=end_dt%20%3D%20start_dt%20%2B%20timedelta,commit), then generates fresh. So if we do this for a specific week, it will completely reshuffle that week’s assignments, not just fill the one gap. This could be disruptive (but ensures coverage). - Another approach: mark the PTO-approved employee as inactive temporarily and call generate for just the period – since the algorithm sorts and chooses deterministically, others will fill in.

We have to weigh complexity vs. benefit. For now, a practical plan: - **Integrate PTO into generation (prevention)** – *this we must do*[[74]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). - **Minimal reshuffle on approval** – perhaps simply call generate for that week and accept that it reassigns shifts (the users should be informed or maybe this action is done by the manager intentionally after reviewing). - Optionally, log or highlight that a manual adjustment was made (maybe via shift trade system, but that’s more manual).

In summary, after these changes, the system will **honor time-off requests**: People on PTO won’t be scheduled in newly generated schedules, and any last-minute PTO approvals can trigger the scheduling algorithm to plug gaps (making use of the “AI” to solve conflicts rather than leaving it entirely to human intervention).

### 5. Scheduling Algorithm Limitations

The greedy scheduling algorithm in place covers many rules from the requirements, but a few issues need attention to fully meet the spec:

* **Minimum Coverage vs. Staffing**: The requirement is minimum 3 on duty at all times[[77]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). The code enforces 3 at night, but uses 4 for day shifts[[78]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shifts%20while%20enforcing%20the%20following,rules). This isn’t necessarily wrong – it could be that daytime requires an extra person (maybe the lead is a supervisor not counted in the 3, or they wanted overlap for shift change). It does ensure at least 3 (since 4 >= 3). We should confirm if having 4 day shift staff is intended or if it should also be 3. If the extra person is not required, we could reduce required\_day to 3 to exactly meet the minimum. However, having 4 might be a deliberate choice to use all 11 employees (e.g., 4 day, 3 night = 7 per day, which over a week gives everyone roughly 40 hours). We’ll keep it at 4 unless directed otherwise, but it’s worth noting this discrepancy.
* **Consecutive Days Off**: The requirements document states *“Consecutive Days: No maximum limit”*[[79]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g), but the code imposes a 5-day max (which is a typical labor rule). This is a slight deviation – possibly the requirements left it open, and the developer added a 5-day cap for safety/fairness. Since the UI even collects “Max Consecutive Work Days” (default 5) per employee, we’ll preserve this rule. It likely improves the schedule by giving everyone breaks, even if not explicitly mandated.
* **Weekly Hour Limit Reset:** A critical bug – the algorithm **does not reset hours\_assigned after each week** when generating multiple weeks in one go. If the user requests 2 or 4 weeks of schedule generation at once, the state dictionary carrying hours\_assigned per employee will continue accumulating hours across weeks. For example, after week 1, many employees will hit ~40 hours; in week 2 of the loop, those employees will appear “at cap” and the scheduler will avoid assigning them additional shifts, leading to an understaffed second week. The code currently computes total\_days = weeks \* 7 and loops through that many days continuously[[80]](file://file_000000009b7461f79961f11bd0037eef#:~:text=start_dt%20%3D%20today%20%2B%20timedelta,Generate%20new%20schedule) without any logic to reset weekly hours. This means a 2-week schedule generation could wrongly treat the second week as overtime for everyone and leave shifts empty (or filled only by those who had hours remaining).

**Fix:** Modify generate\_schedule to handle multi-week generation properly. We can do this by breaking the loop into week-long chunks: - Simple approach: call the generator week by week (e.g., in the endpoint, loop for each week and call generate\_schedule for 7 days at a time, committing each week). But since the current function already deletes the range up front, we might need to adjust that logic if we call it multiple times. - In-function approach: Detect when a new week starts (perhaps track the day of week since start). The algorithm knows Monday because if start\_date is a Monday (which it is by default or by user choice), then every 7th day is Monday again. We can check if day\_idx % 7 == 0 and day\_idx != 0: to signify a week boundary, and then reset each employee’s hours\_assigned = 0 (and maybe also reset days\_worked = 0 if we assume a day off between weeks, though the consecutive-days rule already prevents >5 and likely ensured a break).

Implementing the weekly reset will ensure that each week every employee can work up to their max hours fresh. This aligns with the idea of “40 hours per week” rather than per schedule run[[81]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g).

* **Fair Overtime Distribution:** The requirements mention *“Rotate overtime opportunities fairly among eligible employees”*[[82][83]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). The current implementation doesn’t explicitly do fairness tracking; it sorts employees alphabetically (with lead and nights-only given priority positions) and then greedily assigns shifts. This could lead to the same employees getting overtime regularly if they appear earlier in sorting or have more availability. A potential improvement is to rotate the order of candidates each week or track who got overtime last time. However, doing this properly may require storing history or using a more sophisticated solver. For now, a reasonable tweak is:
* Randomize or rotate the starting order of the employees\_sorted list for each new week or schedule run. Since it’s deterministic now, introducing a rotation could balance out who is first in line for extra shifts over multiple runs.
* Alternatively, implement a simple round-robin for overtime: e.g., when filling that last slot that causes overtime, pick the employee who has the least overtime hours so far. This is not currently tracked, so it may be complex to add without extensive changes.

This fairness aspect is a “nice-to-have” for MVP. We should mention it in the code comments as a future enhancement. The structure of the code is such that adding a more advanced optimization algorithm later (e.g., using linear programming or an external AI scheduler) is possible[[84]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee%20weekly%20caps%2C%20lead%20assignments%2C,in%20a%20more%20sophisticated%20solver), but not required immediately.

* **Special Cases & Preferences:** The requirements list a special employee who works 4-hour shifts, 4 days a week (likely a part-time role)[[85]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g). The current code does not handle 4-hour shifts at all – it assumes all shifts are either ~10.5h or the lead’s 8h. If we need to accommodate a 4-hour shift employee, that’s a new pattern to introduce. We might decide to ignore this for the first production version (the employee could be treated as 20-hour cap and they will just be assigned one full shift which would exceed 4h though – not ideal). If needed, we could create a rule: if an employee’s special\_schedule == "LEGAL\_CAP" or hours\_per\_week < 20, maybe assign them shorter shifts (like 4 or 6 hour slots) specifically. This would complicate the coverage calculations (since four 4-hour shifts would equal one 10.5h shift in coverage time). Given this is an edge case and not currently implemented, it’s acceptable to defer it. Just be aware that the current system doesn’t truly support shifts shorter than the templates defined.

Similarly, *employee preferences* (beyond cannot-work days) aren’t used. The requirements imagined an employee\_preferences table for preferred days etc., but we haven’t implemented that. This again can be a future addition – for now, employees can be given availability via cannot\_work\_days and that’s it. Preferences like “prefers nights” or “prefers overtime” could be factored in later.

In summary, the scheduling algorithm after tweaks will: - Use **employee availability** (no assignment on their off-days or PTO days). - Ensure **coverage** is met (3 min, using 4 in day to be safe). - Honor **lead prioritization** (Patty gets first day shift if available)[[86]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,emp_state%5B%27max_hours)[[87]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_type%3D%27DAY%27%2C%20role%3D%27Lead%27%2C%20is_overtime%3D%28emp_state,hours_assigned%27%5D%20%2B%3D%208). - Respect **nights-only** (Nicole never gets day shifts by construction)[[88]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,is_lead). - Enforce **hour caps weekly** (no one exceeds their weekly max unless overtime is absolutely needed for coverage, flagged as is\_overtime=True on that shift[[89]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_len%20%3D%2012%20is_ot%20%3D,19%2C%200)). - Enforce **rest and consecutive days** (10-hour rest check between shifts[[90]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,5), and no more than 5 days in a row[[91]](file://file_000000009b7461f79961f11bd0037eef#:~:text=available%20%3D%20False%20,hours_assigned%27%5D%20overtime_candidate%20%3D%20False)). - Mark **overtime** shifts (when an assignment pushes hours over the limit) so they can be tracked[[92]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift%2C%20mark%20as%20candidate%20for,True%20if%20not%20available%3A%20continue)[[93]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_len%20%3D%2010.5%20,emp_state%5B%27max_hours). - **Avoid scheduling during PTO** (new integration). - Possibly **reset weekly hours** on multi-week runs (fixing the bug). - Leave room for improvements in fairness and special cases.

We will now outline concrete steps to implement these changes in the code, effectively “wiring up” the backend and enhancing the AI scheduling logic as needed.

## Implementation Plan (Step-by-Step for the AI Agent)

Below is a **task list** to guide the development process. Each step should be executed and tested to ensure the system works end-to-end:

1. **Fix Flask Route Names:** In app.py, rename the duplicate route function for /dashboard. For example, change:

* @app.route('/dashboard')  
  def root\_page(): ...
* to
* @app.route('/dashboard')  
  def dashboard(): ...
* Adjust any references to request.endpoint == 'dashboard' in templates accordingly. Ensure the index route (/) still uses root\_page (or rename it to index() for clarity) so both routes are registered separately[[36]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app). Verify the app starts without route conflicts and that navigating to "/" and "/dashboard" both work.

1. **Align Frontend Field Handling with Backend:**
2. Modify the **Employee POST** handler to read UI fields:
   * Parse data['shift\_type']. Based on its value, set is\_lead or nights\_only:
   * If shift\_type == "DAY": nights\_only = False (and perhaps set a day\_only = True flag – you can implement this by not adding the employee to night shifts in the algorithm, see step 9).
   * If shift\_type == "NIGHT": nights\_only = True.
   * If shift\_type == "BOTH": nights\_only = False.
   * Parse data.get('special\_schedule'). If it's "LEAD", set is\_lead = True. Otherwise, is\_lead stays as provided (or False by default). You may also want to store special\_schedule in the Employee model (add a column) to keep track, but the crucial part is marking leads.
   * Accept hours\_per\_week: use data.get('hours\_per\_week') if available, and assign that to max\_hours\_per\_week. If not provided, default to 40. (Alternatively, change the frontend to send max\_hours\_per\_week directly. Either way works.)
   * Add new fields in the Employee model for max\_consecutive\_days and min\_rest\_hours (both integers). Initialize them with defaults (e.g., 5 and 10) if not provided, and parse the form values to set them. Update the database schema (since this is a development phase, you can drop and recreate or use a migration tool). Also adjust the Employee.to\_dict() to include these new fields (if we plan to use them in UI).
   * Ensure cannot\_work\_days is handled correctly: if the incoming value is a list, do json.dumps(list); if it's already a JSON string, you might double-dump. Ideally, adjust the JS to send an array of days (not stringified) and then do employee.cannot\_work\_days = json.dumps(data.get('cannot\_work\_days', [])) as currently written (this will work if data['cannot\_work\_days'] is a list). Test that in the DB the stored cannot\_work\_days looks like ["Mon","Fri"] for someone who cannot work Mon/Fri, and that json.loads in the generator yields a Python list.
3. **Update Response:** Change the return of employee creation to include a success flag[[45]](file://file_000000009b7461f79961f11bd0037eef#:~:text=db,error%27%3A%20str%28e%29%7D%29%2C%20400). For example:

* return jsonify({"success": True, "employee": employee.to\_dict()}), 201
* Do similarly for time-off creation:
* return jsonify({"success": True, "request": req.to\_dict()}), 201
* (This will satisfy the front-end checks[[44]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX)[[46]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH).)

1. **Edit Employee PUT:** Modify the update\_employee endpoint to handle shift\_type and special\_schedule if they come in the JSON. For instance, if data.get('shift\_type') exists, set employee.nights\_only accordingly (and possibly handle day-only case). If special\_schedule in data, set employee.is\_lead accordingly. Also handle hours\_per\_week -> max\_hours\_per\_week. Update max\_consecutive\_days and min\_rest\_hours if provided. Essentially mirror the logic from creation. This way, edits via API can update those attributes.
2. **Employee DELETE:** No changes needed on backend (already sets inactive). But after calling it from frontend, you might have the backend return something like {"success": True} for consistency.
3. **Adjust Frontend JS for Employees:**
4. In employees.html script, implement the API calls for **edit and delete**:
   * For edit: inside the submit handler for #edit-employee-form (currently stubbed)[[50]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX), collect the fields from the form similar to how the add form does. Construct a JSON object. You might include fields like name, email, shift\_type (deduce from the dropdown which likely has the same options), hours\_per\_week, special\_schedule (from the dropdown), active status, etc. Use apiRequest('/api/employees/'+employeeId, { method:'PUT', body: JSON.stringify(data) }). On success (response.success), show success alert, close modal, and call loadEmployees() to refresh the table. On failure, alert error.
   * For delete: in removeEmployee(id), instead of immediately showing an alert, call apiRequest('/api/employees/'+id, { method:'DELETE' }). On a 200 success, you can show a "removed" alert and refresh the list. If an error comes back (e.g., 404 if employee not found), show a failure alert. Maintain the confirmation prompt as is for safety[[94]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX).
5. Update the **add employee** form submission success handling: after receiving the response from POST, the code currently checks if(response.success). Now that the backend will send {"success": true, "employee": {...}}, this will pass. However, note that our current frontend code doesn’t use the returned employee data at all – it just refreshes the whole list. That’s fine. Optionally, we could optimize by appending the new row without reloading everything, but not necessary.
6. Double-check that the **status badge** logic in displayEmployees correctly reflects active vs inactive[[95]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX). It uses emp.active ? 'Active':'Inactive'. Since our GET still filters out inactive, we won’t see inactive entries at all. We might consider changing GET /api/employees to return all employees (or add a separate endpoint) so that maybe an admin can see inactive staff. But not required now.
7. **Test Employee Management End-to-End:** At this point, test manually (or via automated tests if available):
8. Add an employee via the UI. Ensure no error alert appears, and the new employee shows up in the table. Verify the values (especially shift type and hours/week) are correct. Check the database if possible to see that is\_lead and nights\_only set as intended, etc.
9. Edit an employee’s attributes (make a non-lead into a lead or change their shift preference). After saving, verify the table updated and the DB reflects the change.
10. Delete an employee. They should disappear from the list. Try adding them again (if same name/email) to ensure the system can handle duplicates or reuse (our DB might throw IntegrityError if email is unique – in model it’s not marked unique, but email duplicates might be undesirable; anyway).
11. Check that UI status badges for Active/Inactive still make sense (since inactive are filtered out, probably all we see is "Active" for everyone in list).
12. **Integrate PTO into Schedule Generation:**
13. Modify generate\_schedule(start\_dt, days) in app.py:
    * Before the main loop, query the TimeOffRequest table for all requests with status 'APPROVED' that overlap the target date range. You can do something like:
    * approved\_requests = TimeOffRequest.query.filter(TimeOffRequest.status=='APPROVED',   
       TimeOffRequest.start\_date < end\_dt,   
       TimeOffRequest.end\_date >= start\_dt).all()
    * (Where end\_dt = start\_dt + timedelta(days=days) as computed in the endpoint.)
    * Build a structure from these requests for quick lookup. For example, a dictionary pto\_by\_emp = {emp\_id: [] of dates} or even a set of specific (emp\_id, date, shift\_type\_allowed) tuples. Because shift\_type could be DAY, NIGHT or BOTH, interpret it:
    * If shift\_type is 'BOTH', they are off for all shifts in that date range.
    * If 'DAY', they should not work day shifts in that range, but could still work nights.
    * If 'NIGHT', vice versa.
    * During the scheduling loop for each day (current\_date):
    * For each employee e when building pools, check if that employee has PTO on this current\_date. If yes, skip them for the corresponding shift type:
      + If their PTO shift\_type is BOTH, skip adding them to any pool (they are completely off).
      + If PTO shift\_type is DAY, skip adding to day\_pool (but still allow in night\_pool if they are not otherwise restricted and it’s not conflicting date for nights).
      + If PTO shift\_type is NIGHT, skip adding to night\_pool for that date.
    * This check can be integrated where we currently check cant\_work = json.loads(e.cannot\_work\_days or '[]') and skip if weekday in that[[75]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,last_shift_end%27%5D%3A%20days_since_last%20%3D%20%28current_date). Right after that, do:
    * # Skip if employee is on approved PTO this day  
      if e.id in pto\_by\_emp:  
       if current\_date in pto\_by\_emp[e.id]:  
       # Determine if we should skip day, night or both  
       pto\_type = ... # the shift\_type for that PTO entry covering current\_date  
       if pto\_type in ('BOTH', 'DAY'):  
       # don't allow in day\_pool  
       allow\_day = False  
       if pto\_type in ('BOTH', 'NIGHT'):  
       # don't allow in night\_pool  
       allow\_night = False
    * Then use allow\_day/allow\_night flags to decide adding to each pool. (Alternatively, pre-compute a set of unavailable employees for the day for each pool.)
    * This will ensure the algorithm simply treats those employees as unavailable on their PTO dates, so it will naturally fill shifts with others (if possible). If coverage is very tight and someone with PTO can’t be replaced (e.g., everyone else is also off or at max), then the result may be under-staffing – we might want to detect that and flag an alert (maybe via the “coverage gap alerts for manager” mentioned in requirements[[96]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g), possibly a future enhancement).
    * Test the generation with a scenario: Mark an employee as having PTO on a date and generate the schedule covering that date. Verify that employee does not appear in any shift on that date in the output JSON.
14. **Automatic Reschedule on PTO Approval (Optional Initial Implementation):**
15. In the approve\_timeoff\_request endpoint, after setting status to APPROVED and committing[[16]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%2F,400%20req.status%20%3D%20%27APPROVED), add logic to call the scheduler for the affected period. For example:

* # After approving the request  
  start = req.start\_date  
  end = req.end\_date  
  # Determine number of days  
  days = (end - start).days + 1  
  generate\_schedule(start, days)  
  # This returns a list of Schedule objects (not yet committed in DB)  
  # We might insert these into the DB carefully.
* This approach is tricky because generate\_schedule as written deletes all schedule entries in the range at the start[[76]](file://file_000000009b7461f79961f11bd0037eef#:~:text=end_dt%20%3D%20start_dt%20%2B%20timedelta,commit). It might remove more shifts than necessary (all shifts in that PTO range, even those not involving the PTO employee). Then it regenerates fresh assignments for that range.
* If we isolate the range to exactly the PTO dates, this essentially reassigns those days completely. That might be acceptable – presumably those days need to be adjusted anyway due to the absence.
* We have to call db.session.bulk\_save\_objects(new\_assignments) and commit to save the new schedule for that range[[14]](file://file_000000009b7461f79961f11bd0037eef#:~:text=Schedule.query.filter%28Schedule.schedule_date%20%3E%3D%20start_dt%2C%20Schedule.schedule_date%20,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments). This effectively “reshuffles” the affected days.
  + Caution: If the PTO is just one shift of the day (DAY vs NIGHT), our current generate\_schedule will wipe both shifts on that day and rebuild them. That means if someone had a night shift that day, they might get replaced too, which is unnecessary churn. A more precise approach would be ideal (only regenerate the affected shift type or find a specific replacement), but implementing that precision is complex. For MVP, regenerating the whole day (both shifts) is simpler and ensures coverage.
  + Another angle: We could detect if a trade can solve it. For example, if someone takes PTO, maybe auto-generate a “shift trade request” where they give up their shift and it’s open for others. However, since we have the algorithm, using it directly is fine.
* Given the time constraints, the brute-force regeneration of the PTO date range is an acceptable solution to ensure no open shifts. We should inform the user (maybe via an alert or notification) that the schedule was updated. This could be done by including in the approve API response something like "reshuffled": true or a message listing how many shifts were affected. The UI could then, for example, refresh the schedule view or show a notice. This is a nice-to-have; initially, we can skip the UI part and just rely on the fact the manager likely knows to check the schedule if they approve PTO.

1. **Testing PTO Reshuffle:** Approve a PTO for a date that already had a schedule. After approval, call GET /api/schedule for that date range and see that the shifts have changed to remove the PTO person. Ensure coverage is still met (the algorithm likely plugged someone else). If no one was available, our algorithm might leave that shift unfilled (since it doesn’t explicitly guarantee fill if all are maxed out – in that case, at least the PTO person is removed which is correct, but there’s a gap which should be addressed by a manager manually).
2. This automated step might be left out of first deployment if it’s risky, but it demonstrates how AI can handle dynamic changes. At minimum, log that a PTO approval may require manual rescheduling if not automated.
3. **Correct the Weekly Hours Reset Bug:**
4. Implement the weekly boundary logic in generate\_schedule. As described, if weeks > 1, either:
   * Simplest: do a loop in the /api/schedule/generate route handler:
   * new\_assignments = []  
     current\_start = start\_dt  
     for w in range(weeks):  
      # generate 7 days from current\_start  
      week\_assignments = generate\_schedule(current\_start, 7)  
      new\_assignments.extend(week\_assignments)  
      current\_start += timedelta(days=7)  
     # Delete existing range then bulk\_save new\_assignments as currently done
   * This way, generate\_schedule is always called with days=7, so internally it won’t span week boundaries and each call naturally resets by starting fresh. However, generate\_schedule as written drops *all* schedules in the range given to it. We already do that outside the loop (the current code deletes all schedules in [start\_dt, end\_dt) once). We would need to adjust to delete per week or simply delete the whole range once at start (which is fine as long as we then generate all weeks one by one and accumulate). Make sure not to call init\_db() or drop tables – just remove the specific rows.

* This approach might require adjusting generate\_schedule to not itself call delete so that we can manage deletions externally. Alternatively, we can refactor generate\_schedule into two parts: one that does the assignment logic and returns Schedule objects without touching the DB, and the endpoint that handles deletion and committing.
* For now, to minimize change, we could trick it by calling generate week by week but still using the existing deletion logic properly: - Compute the overall end\_dt. - Delete schedules in range [start\_dt, end\_dt) as before (single deletion). - Then for each week, call generate\_schedule (modified to **not** delete anything internally, since we handled it). That means adjusting or bypassing the Schedule.query.filter(...).delete() inside generate\_schedule when we are doing multi-week loop externally. - One way: add a parameter to generate\_schedule like clear\_db=False to skip the deletion. - Or remove the deletion from generate\_schedule entirely and do it only in the endpoint function.
  + Alternative: incorporate the reset logic inside generate\_schedule: e.g.
  + if day\_idx > 0 and day\_idx % 7 == 0:  
     # reset weekly counters  
     for e in employees\_sorted:  
     state[e.id]['hours\_assigned'] = 0.0  
     # optionally reset days\_worked = 0 or allow it to carry over if no break occurred
  + And maybe sort the employees again by name for fairness at the new week (or rotate). This approach is a bit hacky but contained. It would allow a continuous generation without external loops.
* The downside: if an employee worked Sunday and Monday, with our 5-day limit rule, resetting days\_worked on Monday would let them potentially work 10 consecutive days spanning the week boundary. To prevent abuse, we might not reset days\_worked – we only reset hours. That means someone could still be considered to have worked previous days which is actually true (if no day off on Sunday-Monday transition). So keep days\_worked continuity, only reset hours\_assigned. This is an edge scenario (someone works Thu, Fri, Sat, Sun, then Monday – our code would have stopped them at 5 already, so Monday wouldn’t happen if Sun was their 5th).
  + The external loop method is clearer and aligns with the idea that each week’s schedule is separate.

1. Implement whichever solution is simpler for now. The key is that if the user chooses 2 or 4 weeks, the employees are able to work up to their max each week, not just once. Test by generating 2 weeks and verifying that, for example, Patty (lead) appears in the second week’s schedule as well (currently she would not, because she’d hit 40h in week1 and then be excluded in week2).
2. **Enhance Fairness (if time permits):**
3. As a quick tweak, shuffle the order of employees\_sorted at the start of each week’s assignment. Since we sort by (not e.is\_lead, not e.nights\_only, name) initially[[97]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,str%2C%20any%5D%5D%20%3D) to ensure a deterministic baseline, we could then do something like:

* # After sorting and before scheduling loop:  
  base\_order = employees\_sorted.copy()  
  for week in range(num\_weeks):  
   if week > 0:  
   # rotate the list by one or randomize a bit  
   base\_order = base\_order[1:] + base\_order[:1]  
   # then use base\_order as the order for that week in the loop...
* This way, a different person starts first each week for assignments like the 10.5h shifts, distributing any surplus or overtime more evenly.

1. This is a simplistic approach to fairness. A more robust solution might require tracking how many extra hours or unfavorable shifts each employee got and then adjusting priority next time – which is out of scope for now. The rotation at least ensures the same individual isn’t always picked first when filling shifts after the lead.

This step is optional, but we mention it as it aligns with the “fair distribution” requirement[[82]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g).

1. **Incorporate “Day Only” Employee Preference:**
2. If we have employees who prefer not to work nights (e.g., perhaps some staff are only comfortable with day shifts even if not officially nights-only), the UI’s "Day Only" selection currently doesn’t map to anything. We can interpret "Day Only" as the person should not be assigned night shifts. One way is to add a boolean field day\_only to Employee or infer it as not nights\_only and special\_schedule != 'LEAD' and shift\_type was 'DAY'.
3. Implement: In the schedule generator, when building night\_pool, exclude those employees who have a “day only” preference. We already exclude nights\_only folks from day\_pool and (implicitly exclude leads from night\_pool in code)[[88]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,is_lead). We can similarly exclude those who are not nights\_only but explicitly day-pref only from night\_pool. This requires knowing that preference:
   * If we added a shift\_type field to Employee, it could be 'DAY', 'NIGHT', 'BOTH'. If shift\_type == 'DAY', skip adding them to night\_pool. If shift\_type == 'NIGHT', we already handle (nights\_only True covers that). If 'BOTH', proceed as normal.
   * Alternatively, if we didn’t add shift\_type to DB, we can derive it from nights\_only and is\_lead combination – but that doesn’t distinguish day-only vs both. So adding a field or using the special\_schedule code might be needed. Since UI already had shift\_type, I lean towards adding a shift\_type column (or reusing the UI value to set nights\_only or some flag).
4. For now, to deliver the expected behavior: treat any employee whose nights\_only is False but who in the UI was marked as "Day Only" as effectively “cannot work nights.” We can maintain an internal set of day-only employees (could piggyback on cannot\_work\_days by adding 'Sun','Mon','Tue','Wed','Thu','Fri','Sat' except those that correspond to nights? But that’s messy).
5. Simpler: add a boolean day\_only or a property prefers\_day = True if shift\_type was "DAY". Then in generator:

* if hasattr(e, 'prefers\_day') and e.prefers\_day:  
   # skip them from night\_pool even if not nights\_only  
   # (They will still be in day\_pool)  
   pass
* Actually, code currently adds every non-nights\_only to night\_pool[[98]](file://file_000000009b7461f79961f11bd0037eef#:~:text=day_pool.append%28e%29%20,append%28e), except leads. We can insert a check there:
* if not e.is\_lead and not getattr(e, 'prefers\_day', False):  
   night\_pool.append(e)
* And ensure that for those marked day-only, prefers\_day is True.

1. Implement this if we decide to honor the "Day Only" selection from UI. If not implemented, a user selecting "Day Only" would still be assigned night shifts since the code treats them as available for both. That could be considered a bug if noticed.
2. Testing: Mark an employee as Day Only via the UI, generate schedule, and confirm that person has no Night shift assignments in the output.
3. **Implement Shift Trade Functionality:**
   * **Model & Migration:** Define a ShiftTrade model in app.py:
   * class ShiftTrade(db.Model):  
      \_\_tablename\_\_ = 'shift\_trades'  
      id = db.Column(db.Integer, primary\_key=True)  
      requesting\_employee\_id = db.Column(db.Integer, db.ForeignKey('employees.id'), nullable=False)  
      target\_employee\_id = db.Column(db.Integer, db.ForeignKey('employees.id'), nullable=False)  
      original\_schedule\_id = db.Column(db.Integer, db.ForeignKey('schedules.id'), nullable=False)  
      trade\_schedule\_id = db.Column(db.Integer, db.ForeignKey('schedules.id'), nullable=False)  
      trade\_reason = db.Column(db.Text, nullable=True)  
      status = db.Column(db.String(16), default='PENDING') # PENDING, APPROVED, DENIED  
      requested\_at = db.Column(db.DateTime, default=datetime.utcnow)  
      approved\_at = db.Column(db.DateTime, nullable=True)
   * (Add relationships if needed for convenience, or just join manually in queries.) Run db.create\_all() to add the table (our init\_db() will handle it next time it’s called since we extended the models).
   * **API Endpoints:**
   * GET /api/trades – returns a list of trade requests. We can support a filter by status (similar to timeoff) via query param, or just return all for now. Ensure it joins or includes employee names and shift info, because the UI wants to display names and shift details. We might do a join query or simply post-process: for each trade, fetch Employee names and the two Schedule entries to get shift date/type. For efficiency, possibly craft a query or add properties on the model.
   * POST /api/trades – create a new trade request. Expect JSON with requesting\_employee\_id, target\_employee\_id, original\_schedule\_id, trade\_schedule\_id, and trade\_reason. Validate that:
     + The two schedule IDs are valid and correspond to shifts of those employees respectively (to avoid nonsense requests). Possibly ensure that original\_schedule.employee\_id == requesting\_employee\_id and trade\_schedule.employee\_id == target\_employee\_id (they must be trying to swap those shifts).
     + Optional: ensure the shifts are of the same general time (maybe not required, one could trade a day for a night shift theoretically if both agree). If validation passes, insert a ShiftTrade with status PENDING. Return {"success": True, "trade": trade.to\_dict()} (you’d define to\_dict to include human-readable info too if needed).
   * PUT /api/trades/<id>/approve (and similar /deny) – find the trade, set status to APPROVED or DENIED, set approved\_at time. If approving:
     + Perform the actual swap: retrieve the two Schedule entries:
     + sched1 = Schedule.query.get(trade.original\_schedule\_id)  
       sched2 = Schedule.query.get(trade.trade\_schedule\_id)
     + Swap their employee\_id values:
     + emp1 = sched1.employee\_id  
       emp2 = sched2.employee\_id  
       sched1.employee\_id = emp2  
       sched2.employee\_id = emp1
     + This reassigns the shifts. We should also consider swapping any other identifying info if needed (e.g., if we want to keep a record that these were traded, perhaps set is\_coverage=False or something – but not necessary, a trade is basically just a reassignment). After swapping, save the schedules (commit DB).
     + Mark trade as approved and save.
     + Return {"success": True, "trade": trade.to\_dict()}. If denying, we just mark denied and do not modify any schedule.
   * Note: We should handle potential errors: if the trade was already processed or if schedules are not found. For example, if a trade is approved twice, second time we should 400 out. Similar to how timeoff approve endpoint checks req.status != 'PENDING'[[99]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20not%20req%3A%20return%20jsonify%28,success%27%3A%20True%2C%20%27request%27%3A%20req.to_dict).
   * **Frontend Integration:**
   * In shift\_trades.html JS, replace the mock data usage:
     + On page load, call loadShiftTrades() which will do shiftTrades = await apiRequest('/api/trades') (GET). Then call displayShiftTrades() to render them. The server should send back objects with fields like requesting\_employee\_name, target\_employee\_name, original\_shift (perhaps as date + shift code string), target\_shift similarly, reason, status, requested\_at.
     + You may need to construct those string fields on the backend or send raw data and construct on front. For simplicity, we can have the backend’s to\_dict() for ShiftTrade gather the needed info:
       - requesting\_employee\_name from Employee table (join or attribute if relationship set up).
       - target\_employee\_name.
       - original\_shift and target\_shift as formatted like "YYYY-MM-DD Day D1" etc. We can get this by accessing the Schedule entries via relationships or querying them and formatting: e.g., f"{sch.schedule\_date} {'Day' if sch.shift\_type.startswith('DAY') else 'Night'} {sch.role}".
       - Or, to minimize joins, we could store a copy of those details in the ShiftTrade when creating (not normalization best practice, but easier for quick retrieval). However, it's cleaner to compute on the fly for display.
     + Adjust displayShiftTrades() in JS to use the fields returned by the API, rather than the mock structure. The current template of how it builds rows[[100]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) is fine if our JSON matches those keys.
     + Implement approveTrade(id) and denyTrade(id) to call the API endpoints. On success, we can either refresh the trades list or optimistically update the table. For simplicity, just do loadShiftTrades() again to refresh.
     + Implement the population of the shift dropdowns:
     + When the user selects their name in #requesting-employee dropdown, we should fetch their shifts. We can reuse the schedule data already loaded on the dashboard or make a new call. Perhaps easiest: call /api/schedule?start\_date=<today>&days=7 or for a longer range if needed (maybe allow trades for the next 7 days or 14 days schedule only). This returns all shifts in that range. Filter those where employee\_id == requestingEmployeeId to populate the "Shift to Trade" select, and those where employee\_id == targetEmployeeId for the "Their Shift" select (when target is chosen).
     + Alternatively, create dedicated endpoints like /api/employees/<id>/schedule?start=... for convenience. But that might be overkill.
     + Since this is an agent task, we can implement a quick method: after both employees are selected (or when each is selected), fetch schedule data and filter.
     + We must ensure the user picks two distinct employees (the form UI prevents selecting same in Target by a check in JS already[[101]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj)).
     + The form should probably prevent submission until both shifts are selected. We have required on those selects, so that’s fine.
     + Remove or comment out the parts of JS that were faking local approval/denial (the code from line 356 onward in shift\_trades.html that updated the shiftTrades array without API)[[102][103]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj), since we’ll now rely on server responses to update the UI.
     + Also remove the local addition of newTrade in the form submit handler (lines 429-445)[[104][105]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj). Instead, on successful POST response, we can simply show success and refresh the list by calling loadShiftTrades() again, which will include the new request.
   * After these changes, test a full cycle:
     1. Using the UI, submit a new trade request (pick two employees and their respective shifts to swap). Verify it appears in the Shift Trades table as PENDING.
     2. Approve that trade via the UI. The table entry should update to Approved (and possibly disappear if filtering by status). Check the Schedule page or Dashboard – the shifts for those two employees on those dates should now be swapped. For example, if Alice gave Bob her Day shift on 10/12 and Bob gave Alice his Night shift on 10/13, after approval, Alice should appear on the night of 10/13 schedule and Bob on the day of 10/12.
     3. Try denying a trade and ensure it updates accordingly and no schedule changes occur.
     4. Try edge cases: requesting a trade where one employee is not actually on the shift (the backend validation should catch this and return an error; ensure the UI handles error by alerting the user). Also ensure the dropdowns only list valid shifts so such a case is rare.
4. **Finalize and Document:** Clean up any debug prints, ensure all new changes are integrated. Update the README or documentation to reflect how PTO is handled and how to use the shift trade feature. Mention any assumptions (like needing to regenerate schedule after major changes, etc.). This will help the end user or any manager using the system to understand the automation.

Throughout the implementation, keep the requirements in mind to verify each point is addressed:

* We now ensure **time-off requests are respected in scheduling**, meeting the “Process approved time-off” step[[74]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g).
* The **minimum staffing** rule (3 per shift) is met or exceeded, and any under-coverage should be noticeable (perhaps log a warning if after scheduling, any day has <3 scheduled).
* The **lead employee priority** is implemented (Patty gets day shifts)[[86]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,emp_state%5B%27max_hours), and now the UI can actually mark someone as lead so that this rule triggers.
* **Night shift only** employees will never get day shifts (already enforced) and **day-preference** employees won’t get nights (newly enforced).
* **Overtime distribution** is slightly improved by rotation, though a more advanced fairness mechanism could be added in v2.
* The **shift trade system** will allow resolving conflicts or changes without full regen (especially for scenarios not handled by the generator, like two employees wanting to swap shifts). This addresses a requirement for shift trade requests[[106]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g).
* **Employee management** (CRUD) fully works, which is fundamental for real use.

Finally, double-check cross-page integration: e.g., if an employee is made inactive, the schedule generator should skip them (since GET /api/employees filters them out, the generator only loads active employees[[62]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employees%3A%20List,emp.to_dict%28%29%20for%20emp%20in%20employees), so that’s handled). If a shift trade is approved, it might schedule someone beyond their original hours – our algorithm constraints aren’t applied here, so a trade could technically make someone go over 40 hours or break the 10h rest rule if we’re not careful. We might need to validate a trade request against such rules (for example, if approving a trade would cause an employee to have two shifts in one day or over 5 in a row, maybe the system should reject it). This complexity can be addressed in future improvements. For now, assume trades are manager-approved and they will avoid obvious rule breaks.

## Conclusion

By performing the above fixes and enhancements, we connect all the pieces of the Scheduler\_App codebase into a coherent, working system:

* **Backend functions** will properly create, update, and remove records, sending appropriate responses that the frontend expects.
* The **frontend UI** will correctly reflect changes and allow the user (scheduler) to perform all necessary actions without encountering non-functional buttons or misleading error messages.
* The **scheduling “AI”** algorithm will handle real-world constraints more robustly: no scheduling someone who is on PTO, maintaining coverage despite absences, and obeying weekly hour limits and rest requirements. In the event of changes (like a sudden PTO), the system can intelligently reshuffle assignments (or facilitate a trade) to fill gaps, rather than leaving all conflict resolution to the user.
* We have also set the stage for future improvements (like more complex fairness logic, support for custom shift lengths, integration with notifications, etc.) by structuring the code in a clear way and utilizing the data from requirements.

With these changes, the Hospital PBX Scheduling System should be **close to production-ready**, fulfilling the core MVP features from the requirements document[[107]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) and providing a stable platform for scheduling 11 employees 24/7 with automated conflict resolution. The next steps would be thorough testing with real data and perhaps UI refinements, but the heavy lifting of making the code actually work is addressed by the above plan.

[[1]](file://file_000000009b7461f79961f11bd0037eef#:~:text=class%20Employee%28db.Model%29%3A%20,in%20the%20scheduling%20system) [[2]](file://file_000000009b7461f79961f11bd0037eef#:~:text=class%20TimeOffRequest%28db.Model%29%3A%20,request%20submitted%20by%20an%20employee) [[3]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20__name__%20%3D%3D%20%27__main__%27%3A%20,run%28debug%3DTrue%2C%20port%3D5005) [[4]](file://file_000000009b7461f79961f11bd0037eef#:~:text=Maximum%20number%20of%20hours%20the,datetime%20Timestamp%20of%20creation) [[5]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27employees%27%20id%20%3D,Boolean%2C%20default%3DTrue) [[6]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27schedules%27%20id%20%3D,Lead%2C%20D1%2C%20N1) [[7]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,schedule_date%3Dcurrent_date%2C%20shift_start%3Dshift_start%2C%20shift_end%3Dshift_end%2C%20shift_type%3D%27DAY%27%2C%20role%3Drole_label) [[8]](file://file_000000009b7461f79961f11bd0037eef#:~:text=__tablename__%20%3D%20%27timeoff_requests%27%20id%20%3D,Text%2C%20nullable%3DTrue) [[9]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%27%2C%20methods%3D%5B%27GET%27%5D%29%20def%20get_employees%28%29%20,list%20of%20all%20active%20employees) [[10]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,try%3A%20if%20%27name%27%20in%20data) [[11]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Fschedule%27%2C%20methods%3D%5B%27GET%27%5D%29%20def%20get_schedule%28%29%20,for%20a%20given%20date%20range) [[12]](file://file_000000009b7461f79961f11bd0037eef#:~:text=schedules%20%3D%20Schedule.query.filter%28%20Schedule.schedule_date%20,sch.to_dict%28%29%20for%20sch%20in%20schedules) [[13]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Fschedule%2Fgenerate%27%2C%20methods%3D%5B%27POST%27%5D%29%20def%20generate_schedule_endpoint%28%29%20,for%20a%20given%20date%20range) [[14]](file://file_000000009b7461f79961f11bd0037eef#:~:text=Schedule.query.filter%28Schedule.schedule_date%20%3E%3D%20start_dt%2C%20Schedule.schedule_date%20,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments) [[15]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%27%2C%20methods%3D,upper) [[16]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Ftimeoff%2F,400%20req.status%20%3D%20%27APPROVED) [[17]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,contiguous%20range%20of%20days) [[18]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,when%20necessary%20to%20meet%20coverage) [[19]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shifts%20while%20enforcing%20the%20following,rules) [[20]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,when%20necessary%20to%20meet%20coverage) [[21]](file://file_000000009b7461f79961f11bd0037eef#:~:text=db.session.commit%28%29%20,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments) [[36]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app) [[39]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app) [[41]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee%20%3D%20Employee%28%20name%3Ddata,%29%20db.session.add%28employee) [[42]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20%27nights_only%27%20in%20data%3A%20employee,to_dict%28%29%29%20except%20Exception%20as%20e) [[43]](file://file_000000009b7461f79961f11bd0037eef#:~:text=is_lead%3Dbool%28data,get%28%27cannot_work_days%27%2C) [[45]](file://file_000000009b7461f79961f11bd0037eef#:~:text=db,error%27%3A%20str%28e%29%7D%29%2C%20400) [[47]](file://file_000000009b7461f79961f11bd0037eef#:~:text=reason%3Dreason%2C%20%29%20db,to_dict%28%29%29%2C%20201) [[48]](file://file_000000009b7461f79961f11bd0037eef#:~:text=new_assignments%20%3D%20generate_schedule%28start_dt%2C%20total_days%29%20db,success%27%3A%20True%2C%20%27generated_shifts%27%3A%20len%28new_assignments) [[51]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,name) [[52]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee.nights_only%20%3D%20bool%28data,to_dict%28%29%29%20except%20Exception%20as%20e) [[54]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app.route%28%27%2Fapi%2Femployees%2F,404%20employee.active%20%3D%20False) [[56]](file://file_000000009b7461f79961f11bd0037eef#:~:text=nights_only%3Dbool%28data,) [[57]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,if%20weekday%20in%20cant_work%3A%20continue) [[60]](file://file_000000009b7461f79961f11bd0037eef#:~:text=data%20%3D%20request,is_lead%27%5D%29%20if%20%27nights_only%27%20in%20data) [[61]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,status%27%3A%20%27success) [[62]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employees%3A%20List,emp.to_dict%28%29%20for%20emp%20in%20employees) [[66]](file://file_000000009b7461f79961f11bd0037eef#:~:text=%40app) [[75]](file://file_000000009b7461f79961f11bd0037eef#:~:text=for%20e%20in%20employees_sorted%3A%20,last_shift_end%27%5D%3A%20days_since_last%20%3D%20%28current_date) [[76]](file://file_000000009b7461f79961f11bd0037eef#:~:text=end_dt%20%3D%20start_dt%20%2B%20timedelta,commit) [[78]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shifts%20while%20enforcing%20the%20following,rules) [[80]](file://file_000000009b7461f79961f11bd0037eef#:~:text=start_dt%20%3D%20today%20%2B%20timedelta,Generate%20new%20schedule) [[84]](file://file_000000009b7461f79961f11bd0037eef#:~:text=employee%20weekly%20caps%2C%20lead%20assignments%2C,in%20a%20more%20sophisticated%20solver) [[86]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,emp_state%5B%27max_hours) [[87]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_type%3D%27DAY%27%2C%20role%3D%27Lead%27%2C%20is_overtime%3D%28emp_state,hours_assigned%27%5D%20%2B%3D%208) [[88]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,is_lead) [[89]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_len%20%3D%2012%20is_ot%20%3D,19%2C%200) [[90]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,5) [[91]](file://file_000000009b7461f79961f11bd0037eef#:~:text=available%20%3D%20False%20,hours_assigned%27%5D%20overtime_candidate%20%3D%20False) [[92]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift%2C%20mark%20as%20candidate%20for,True%20if%20not%20available%3A%20continue) [[93]](file://file_000000009b7461f79961f11bd0037eef#:~:text=shift_len%20%3D%2010.5%20,emp_state%5B%27max_hours) [[97]](file://file_000000009b7461f79961f11bd0037eef#:~:text=,str%2C%20any%5D%5D%20%3D) [[98]](file://file_000000009b7461f79961f11bd0037eef#:~:text=day_pool.append%28e%29%20,append%28e) [[99]](file://file_000000009b7461f79961f11bd0037eef#:~:text=if%20not%20req%3A%20return%20jsonify%28,success%27%3A%20True%2C%20%27request%27%3A%20req.to_dict) app.py

<file://file_000000009b7461f79961f11bd0037eef>

[[22]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK) [[23]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK) [[24]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK) [[25]](https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK) dashboard.html

<https://drive.google.com/file/d/1krLsqhPgSSJzS1IZIDGOb60uPW6H2_iK>

[[26]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[27]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[28]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[29]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[40]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[44]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[50]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[53]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[55]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[58]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[94]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) [[95]](https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX) employees.html

<https://drive.google.com/file/d/19ORMl7deI30mmk7MY_xSA8oArHzfPuEX>

[[30]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[31]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[32]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[33]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[46]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[49]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[59]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[71]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[72]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) [[73]](https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH) timeoff.html

<https://drive.google.com/file/d/1TMIXVqod01a2vh5bo93PzbjzeKGvNrkH>

[[34]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[35]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[63]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[64]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[65]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[69]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[70]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[100]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[101]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[102]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[103]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[104]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) [[105]](https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj) shift\_trades.html

<https://drive.google.com/file/d/1KXAcV2i34KgMHevxbzZ8n4uN-8zWocOj>

[[37]](https://drive.google.com/file/d/1rY4_AhLweywarRv1dUOTooV21tfe1MVp) [[38]](https://drive.google.com/file/d/1rY4_AhLweywarRv1dUOTooV21tfe1MVp) base.html

<https://drive.google.com/file/d/1rY4_AhLweywarRv1dUOTooV21tfe1MVp>

[[67]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[68]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[74]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[77]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[79]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[81]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[82]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[83]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[85]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[96]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[106]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) [[107]](https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g) hospital-scheduling-requirements.md

<https://drive.google.com/file/d/1sZyuuLRzAjQ1RnK9FTICPR8W5_pWzr8g>