Deliverable: Programmer's Guide Document

Part 1: Project Structure and Stack

High-Level Overview

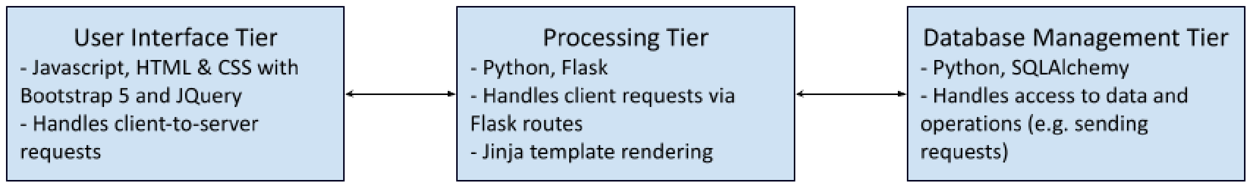
User Interface Tier
- Javascript, HTML & CSS with Bootstrap 5 and JQuery
- Handles client-to-server requests
Processing Tier
- Python, Flask
- Handles client requests via Flask routes
- Jinja template rendering
Database Management Tier
- Python, SQLAlchemy
- Handles access to data and operations (e.g. sending requests)

Figure. A top-level diagram for the three tiers of our project.

Our project splits our python modules into three main tiers: the user interface tier, which includes all of our javascript, html, and css files that we use to compose the front-end of our websites; the processing tier, which is used to process client requests; and the database management tier, which provides all of our functions involving modifications to or reads from the database. The relationships between these high-level tiers is shown above. In particular, any communication between the user interface and the database must traverse the processing tier first, where we can perform security checks before communicating with the database management tier. Within the "gymbuddies" directory, our tiers are organized like so:

- Database management tier: gymbuddies/database
- Processing tier: gymbuddies/*.py
- User interface tier: gymbuddies/static and gymbuddies/templates

In terms of technologies, we use Flask and Python for our server, the Jinja2 template engine to render our HTML, and SQLAlchemy to manipulate and access our database. Our server and Postgres database are hosted on Render. On the front-end, we use Javascript, HTML, and CSS with Bootstrap 5 and JQuery, and we use AJAX to allow clients to send requests to the server with partial refresh. Our front-end is based on the free, open-source Argon Dashboard by Creative Tim[1] and contains a calendar UI by Artsy[2].

Part 2: Database Management Tier

Database Schema

Our database schema involves two main tables: the users and the requests table. A third table called schedules provides an alternative representation of the schedules stored in the users table, and is used to provide access to user schedule information in a more convenient form for the matchmaking algorithm, so that schedule intersections can be found without having to query schedules from every user. A diagram is provided in the figure below.

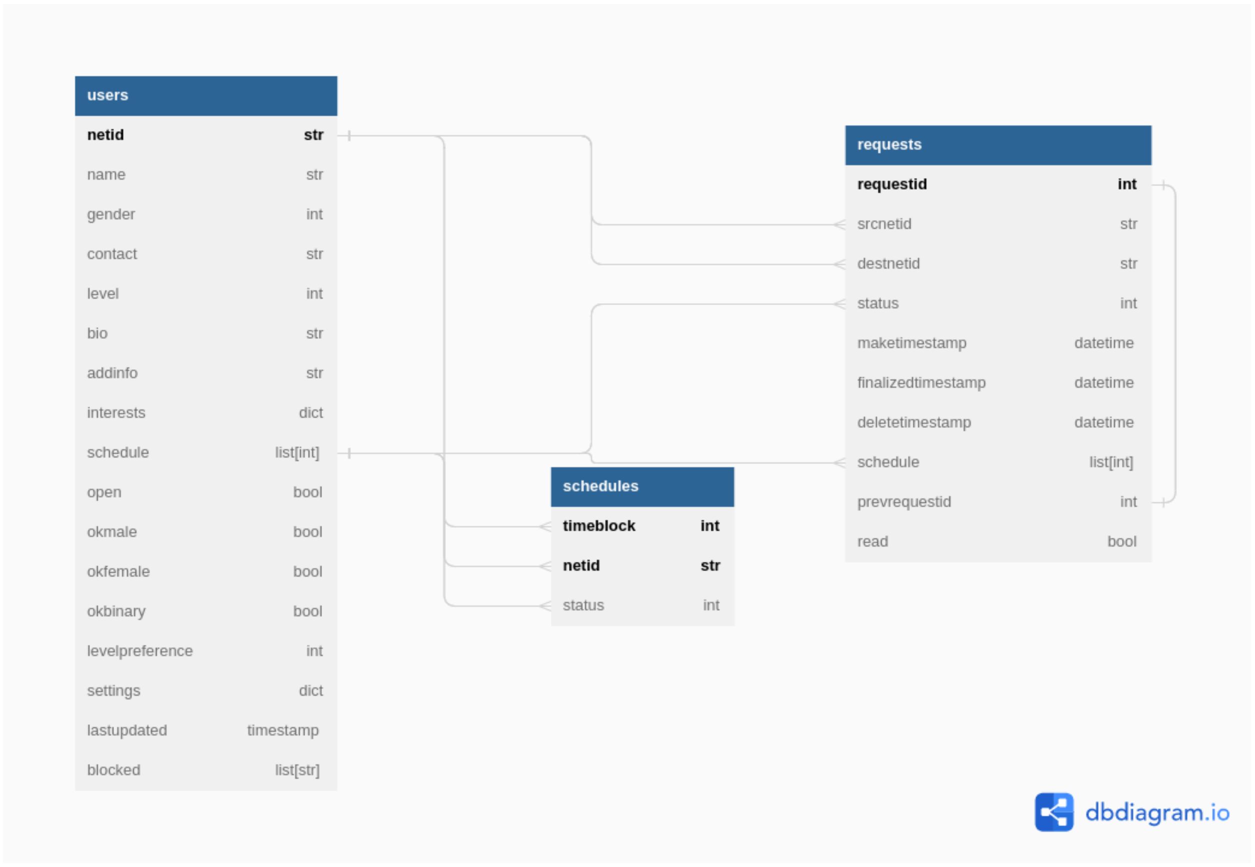

Figure. Our database schema.

Since our application operates within the Princeton ecosystem, netid's can be used as unique identifiers. Hence we use netid's to uniquely identify users in our database. For each user, a row is stored in the users table with:

- profile information and matchmaking preferences of each user (name, contact, etc),
- a settings dictionary,
- a schedule, as a list of integers indicating availability,
- and a lastupdated timestamp.

The "settings" column of a user is a dictionary that currently supports only the "notifications" key, which is True if the user has enabled SMS notifications, and False otherwise.[1] The schedule of a user is stored as a list of integers, whose indices correspond to particular time blocks throughout the week, and whose values are integer flag enumeration with either the AVAILABLE flag or the MATCHED flag, or both. This format is standardized throughout our database, to simplify logic involving schedules. To allow our matchmaking algorithm to find

[1] We chose to store the settings column so that we would not have to worry about database migration whenever we wanted to add a new user setting. By storing the settings column as a dictionary, we can add and remove settings freely.

users whose schedules intersect a particular user' schedule, we maintain the schedules table. For each block in a user's schedule that they are available, we create a row in the schedules table with the appropriate time block (corresponding to an index of a user's schedule) and netid.

The lastupdated timestamp of a user is updated whenever the client makes a change to their profile, or when a request involving them is created or changed. This timestamp is used by the server to decide when to re-process client refresh requests. We perform this timestamp update at the end of a modifying transaction to ensure that when the change in the timestamp is visible to the server, all of the operations in the transaction have already been completed.

Requests made between users are stored in the requests table. A row in the requests table holds:

- a unique, identifying "requestid",
- "srcnetid", the netid of the user who made the request, and "destnetid", the netid of the receiving user,
- status, either REJECTED, PENDING, FINALIZED, or TERMINATED,
- timestamps for the creation, finalization, and deletion of the request,
- a "prevrequestid", the id of the request that was modified into this request, if this was a modification request,
- a schedule with proposed times, and
- a "read" flag, which is used to determine what requests to show in a user's in-app notifications.

There may be multiple requests between the same pair of users in the database, since we keep records of previous requests in the database. When a request is created, its status is initialized as PENDING. From there, it can either be rejected (REJECTED), or accepted (FINALIZED). When a request is finalized, i.e. it becomes a completed match, canceling the request changes its status to TERMINATED, rather than REJECTED. Whenever we finalize or terminate a match, we update the "schedule" column of the user in the users table to be consistent, and similarly we also update the schedules table. This way we differentiate between past matches and past requests which never became matches. The timestamps for these various stages of the request are used to order them in queries. A pending or finalized request may also be "modified" by a user. Modifying a request is almost equivalent to rejecting or terminated it and then sending a new request with different times; the only difference on the database side is that we require the new request to have at least some different times, and we attach the id of the request that was modified to the new request in the "prevrequestid" column. The proposed times of the schedule are stored in the "schedule" column in the same format as the "schedule" column of the users table—the only difference is that the request schedule considers any non-zero value to indicate a selected time.

When making and altering requests and users, we require that the following invariants hold about requests between users:

- There is at most one active request between a pair of users at a time.
- There is no request for which one user has blocked the other.

- If a user is matched with someone for a certain time block, then they have no other active requests for that time block.

The third invariant guarantees that a user cannot accept a request that conflicts with another match involving the source or target user. To enforce this invariant, when a request is accepted, any other pending requests with intersecting proposed times are automatically rejected.

The "read" flag of a request indicates whether or not the request has been read by the appropriate user—for pending requests, the appropriate user is on the receiving end; for finalized requests, the appropriate user is on the source end (since the receiving user accepted the request, they already know that the request is finalized). Thus we only need one "read" flag for each request; the relevant user can be determined via the "status" column. Whenever the status of a request changes, its read flag is marked as False. A pending request is marked as read whenever a user views their pending incoming requests; similarly, a match is marked as read whenever the source user views their matches.

Implementation

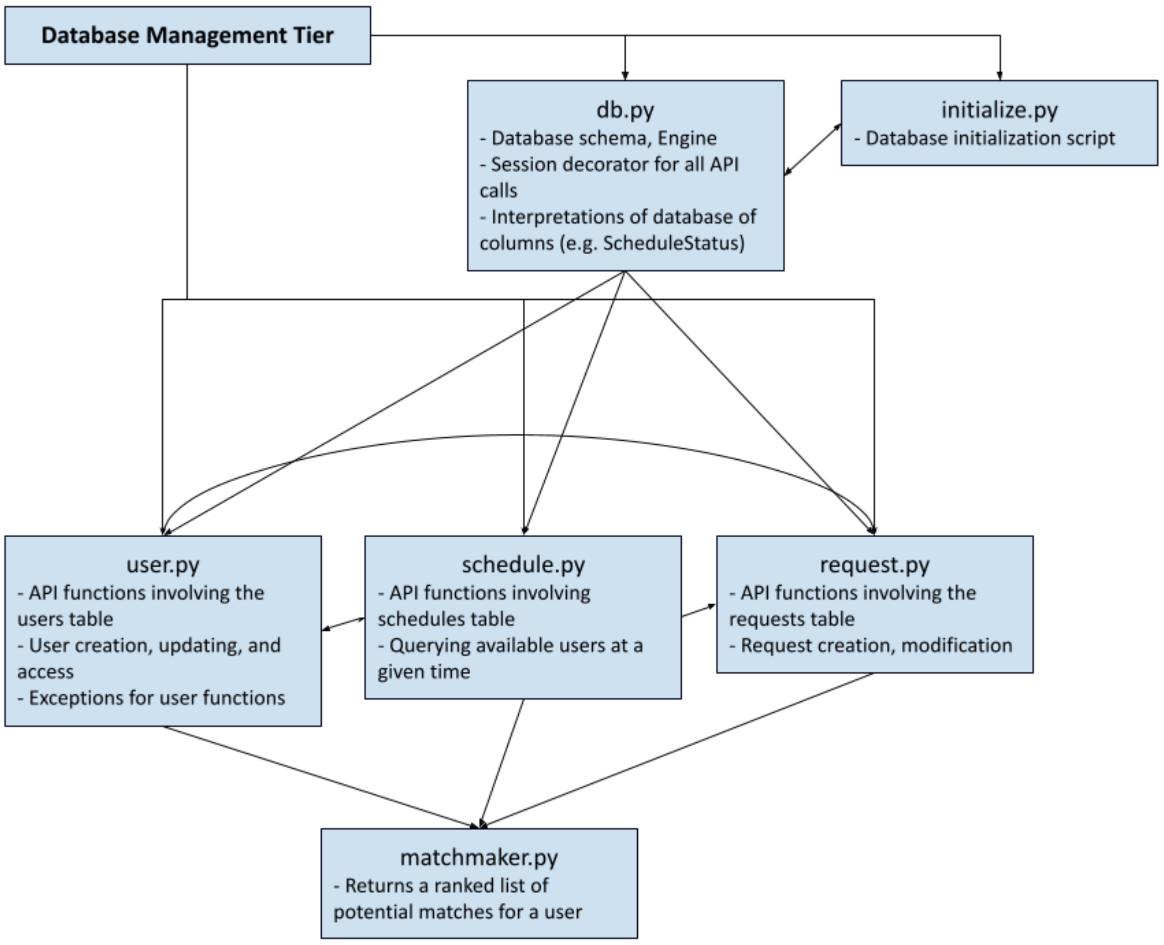
Database Management Tier
db.py
- Database schema, Engine
- Session decorator for all API calls
- Interpretations of database of columns (e.g. ScheduleStatus)
initialize.py
- Database initialization script
user.py
- API functions involving the users table
- User creation, updating, and access
- Exceptions for user functions
schedule.py
- API functions involving schedules table
- Querying available users at a given time
request.py
- API functions involving the requests table
- Request creation, modification
matchmaker.py
- Returns a ranked list of potential matches for a user

Figure. Database management tier module organization.

Our database API functions are implemented in Python using SQLAlchemy. The database metadata is declared in db.py, and the database is dropped and created with the initialize.py script. We split our API functions into four modules: the three basic modules are user.py, schedule.py, and request.py, corresponding to the tables that they handle. The fourth module, matchmaker.py, provides a single API function for querying potential matches of a user. We maintain in our code that direct queries and changes to the users table are only done in user.py, and similarly for schedule.py and request.py. In this way communication between the modules is limited to only what is necessary to enforce consistency among the tables and check invariants (particularly when making requests).

The db.py module handles the SQLAlchemy representation of our database schema, and provides enumerations to interpret the various columns of each table. For instance, the db.RequestStatus class is an Integer wrapper for the "status" column of the requests table, which provides names for the different possible integer values (REJECTED is 0, PENDING is 1, etc). The db.py module is responsible for instantiating a connection to the database via SQLAlchemy's "create_engine" function; this engine is used to create sessions for all of our database API functions. We use a global SERIALIZABLE isolation level, to ensure that we are able to maintain our request invariants and consistency between our tables.

To handle the opening and closing of sessions automatically, db.py provides the "session_decorator" function, a decorator factory which is applied to every exported database API function—i.e. any functions that may be accessed by our server. We use this session decorator to open a session for the duration of a single transaction, and to commit exactly once at the end of write transactions only. Whenever we call a database API function within another database API function, we pass the outer session into the inner call, so that the inner API function does not instantiate another session, or worse perform another commit, breaking our serializability guarantees. The session decorator also handles the updates of the lastupdated timestamp, by collecting actions—in this case, timestamp updates—in the "session.info" dictionary to execute at the end of the transaction.

Error Propagation

Because we use the SERIALIZABLE isolation level, deadlocks sometimes occur when processing request operations. Rather than prevent them,[2] we catch deadlocks in the session decorator, and then timeout for a random period of time before retrying the transaction. Randomizing the timeout makes it likely that one of the competing transactions will start and complete first, resolving the deadlock. In some sense this approach is similar to optimistic concurrency control, but results in more blocking, since all of the deadlocked transactions must

[2] This is complicated by a couple of factors. Attaching sequence numbers to resources to acquire locks in the same order in concurrent transactions is not possible since many of our transactions involve resources unknown at the start of the transaction (e.g. to finalize a request, we have to find all requests that need to be rejected/terminated). Optimistic concurrency control is possible, but requires transaction numbers checks for each transaction.

rollback and retry. This timeout and retry mechanism is also used to retry transactions in the case of unexpected disconnections—which, with the Render database, can happen if a session is left open for too long—or other uncontrollable SQLAlchemy exceptions.

While we catch SQLAlchemy exceptions early and attempt to retry accordingly, we raise errors related to broken database invariants, invalid parameters, and unauthorized operations, so that they can be propagated to the Flask application and handled intelligently there. For each of these exceptions we define a custom exception class, so that the Flask application can identify the exact type of error that occurred.

Matchmaking Algorithm

Our matchmaking algorithm is implemented by the "find_matches" function in matchmaker.py. The algorithm collects a sample of random users from the database, and then filters out users according to certain hard filters. In particular, potential matches should:

- be open,
- match the user's preferred genders,
- have at least one intersecting available time,
- not already have an active request with the user,
- not be blocked by the user or have blocked the user.

The remaining users are analyzed for the intersection of their interests, level preference, and availability, and ranked accordingly. It should be noted that for small databases, we found that the use of the schedules table was not necessary for efficiency; the matchmaking algorithm is already sufficiently fast. Indeed, our current implementation of the matchmaking algorithm does not use the schedules table.

Part 3: Processing Tier

Main Blueprints and Routes

The primary components of our processing are tier three main blueprints—home.py, matching.py, and auth.py—along with the error.py and common.py module, which are shared among the blueprints. The sendsms.py module is a small auxiliary module which is used in matching.py to send out notifications when new requests are made and when requests are finalized. For development environments, we have the master.py blueprint, which provides a route to a master debugger. The master debugger provides a simple GUI to access many of our database API functions, and can be used to test basic functions or query information about users and requests. Our flask application is created by the "create_app" function defined in __init__.py. An overview of the relations between these components is shown in the figure below.

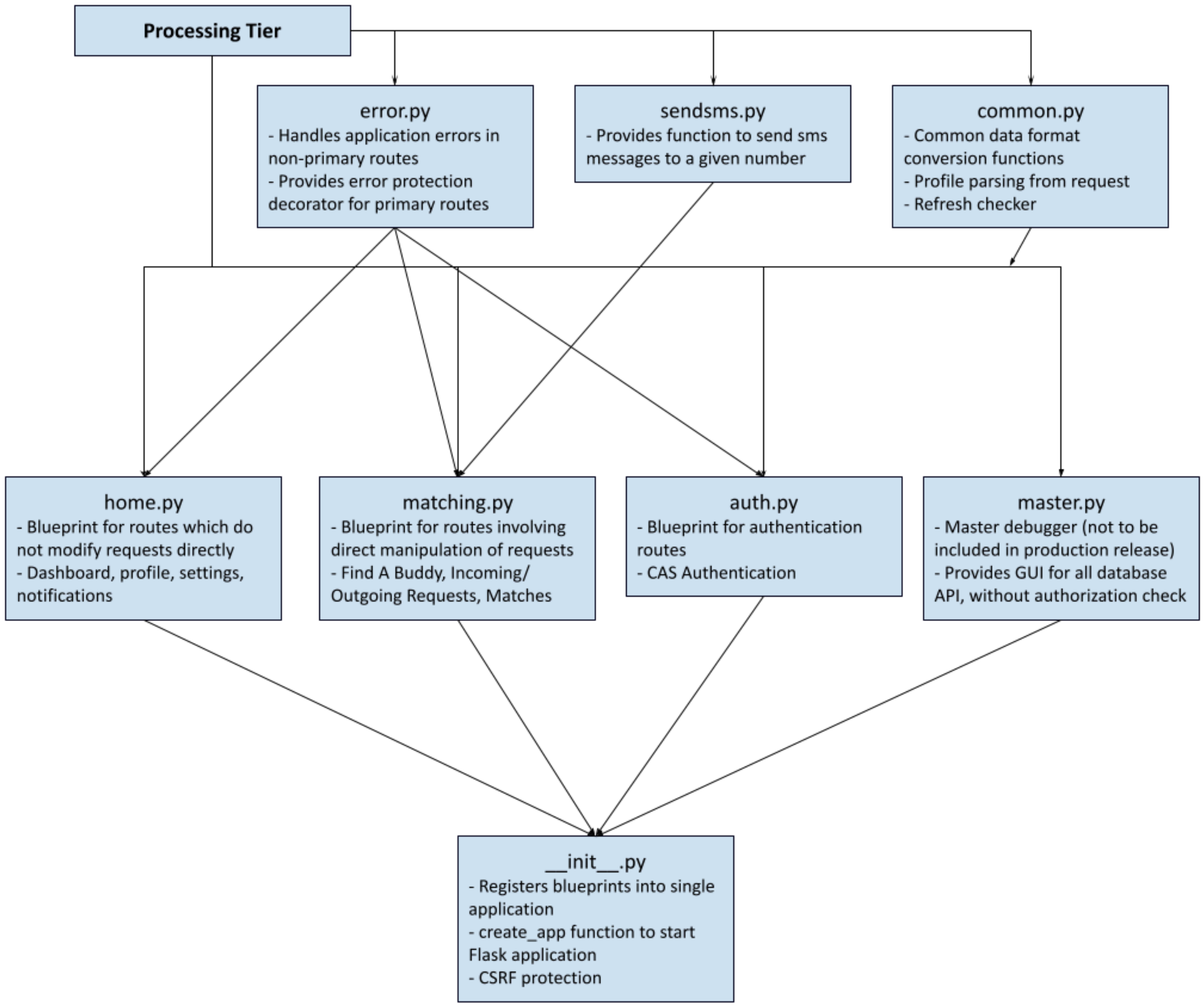
Processing Tier
error.py
- Handles application errors in non-primary routes
- Provides error protection decorator for primary routes
sendsms.py
- Provides function to send sms messages to a given number
common.py
- Common data format conversion functions
- Profile parsing from request
- Refresh checker
home.py
- Blueprint for routes which do not modify requests directly
- Dashboard, profile, settings, notifications
matching.py
- Blueprint for routes involving direct manipulation of requests
- Find A Buddy, Incoming/ Outgoing Requests, Matches
auth.py
- Blueprint for authentication routes
- CAS Authentication
master.py
- Master debugger (not to be included in production release)
- Provides GUI for all database API, without authorization check
__init__.py
- Registers blueprints into single application
- create_app function to start Flask application
- CSRF protection

Figure. Processing tier module organization.

The routes in each of our main blueprint modules can be split into two categories: main view routes, which return templates that render a complete page, and partial view routes, which return templates that render portions of a page. Main view routes that involve are given the "guard_decorator" decorator, which guards against some select errors that can occur when users are deleted—the guard decorator should not be used in partial view routes, since errors in these routes are handled via the Flask application error handlers, registered in error.py. The difference here is that the guard_decorator returns an empty string upon failure, whereas the error handlers return informative json responses. These json responses can be used by the browser to display descriptive error messages, but appear out of place if returned in place of a main view route. To prevent the json responses from appearing to clients directly, we use the guard decorator to catch the error first.

The auth.py Blueprint

The auth.py module handles CAS authentication, but it also provides a CAS_FLAG, which can be turned off to disable CAS authentication and use unauthenticated login via just netid. When CAS is authenticated, the "login" route asks CAS to authenticate the user; if CAS responds successfully, then the user's netid is loaded into the Flask session variable, so that it persists over the browser session. While the session still holds the user's netid, the user is considered logged in, and is able to access routes in the other modules. Each route that requires a logged-in user starts by checking to see that a netid is still stored in the Flask session variable; otherwise, it redirects to the index route. When the user signs out using the "logout" route, it logs out of CAS and then redirects to the index route.

The home.py Blueprint

The home.py module contains routes involving the Dashboard, Profile, and Settings pages. The main page view routes and their children partial view routes (children in the sense that they render only select portions of the main page) in home.py are:

- index (doesn't have a guard decorator since it calls no database API functions)
- profile
    - profilecard: a GET/POST method for rendering an editable version of the user's profile and calendar and updating the user's profile
    - notificationstable: a partial view route for every logged-in page; renders a list of unread notifications for the user
- newuser: first time sign in page for new users
    - profilecard
- tutorial
- dashboard
    - notificationstable
- settings
    - notificationstable
    - blocksearch: a GET/POST route for rendering a block search input and submitting a netid to block
    - blockedtable: a GET/POST route for rendering a list of blocked users and for refreshing the blocked table or unblocking
    - settingsnotifs: a GET/POST route for rendering the notifications toggle and for submitting notification setting changes
    - delete: a POST route for deleting the user's account
- aboutus

When the user submits a profile modification request, we use the "form_to_profile" function from common.py to parse a profile from the request's form, and pass that to the database. For the calendar availability update, this involves receiving a json-encoded calendar from the client and decoding and converting it into a form suitable for the database. Because our calendar UI uses an event-based representation of schedules, where events are specified by their start and end times, we use the "json_to_schedule" conversion function to process the event-based form into a format compatible with the database. Similarly when we send a schedule to the client to render, we convert it into an event-based format using "schedule_to_events" from db.py.

The matching.py Blueprint

The matching.py blueprint handles routes related to making and altering requests, and is responsible for the Find A Buddy, Incoming Requests, Outgoing Requests, and Matches pages. The main view and partial viewroutes of the matching.py module are:

- findabuddy
    - buddies: GET route for rendering the profile and availability calendar of a potential match
- incoming
    - incomingtable: GET/POST route for rendering a list of incoming requests, and for rejecting or accepting a request
    - incomingmodal: GET/POST route for rendering the proposed times and profile of an incoming request, and for modifying requests
- outgoing
    - outgoingtable: GET/POST route for rendering a list of outgoing requests and deleting requests
- matched
    - matchedtable: GET/POST route for rendering a list of active matches, and for terminating existing matches
    - historytable: GET route for rendering a list of past matches
    - unmatchmodal: GET route for rendering a confirmation popup for unmatching

The findabuddy and buddies routes use matchmaker.py's "find_matches" to get a list of matches, and then stores that list of matches into the user's session. To cycle through the matches, we also maintain an "index" variable in the session, indicating which match the user should see next on Find A Buddy; when the user presses "pass" or sends a request, the index automatically increments to show the next match. When there are no more matches, the buddies route renders a prompt for the user to refresh, and then calls "find_matches" again to obtain the next set of matches.

Part 4: User Interface Tier

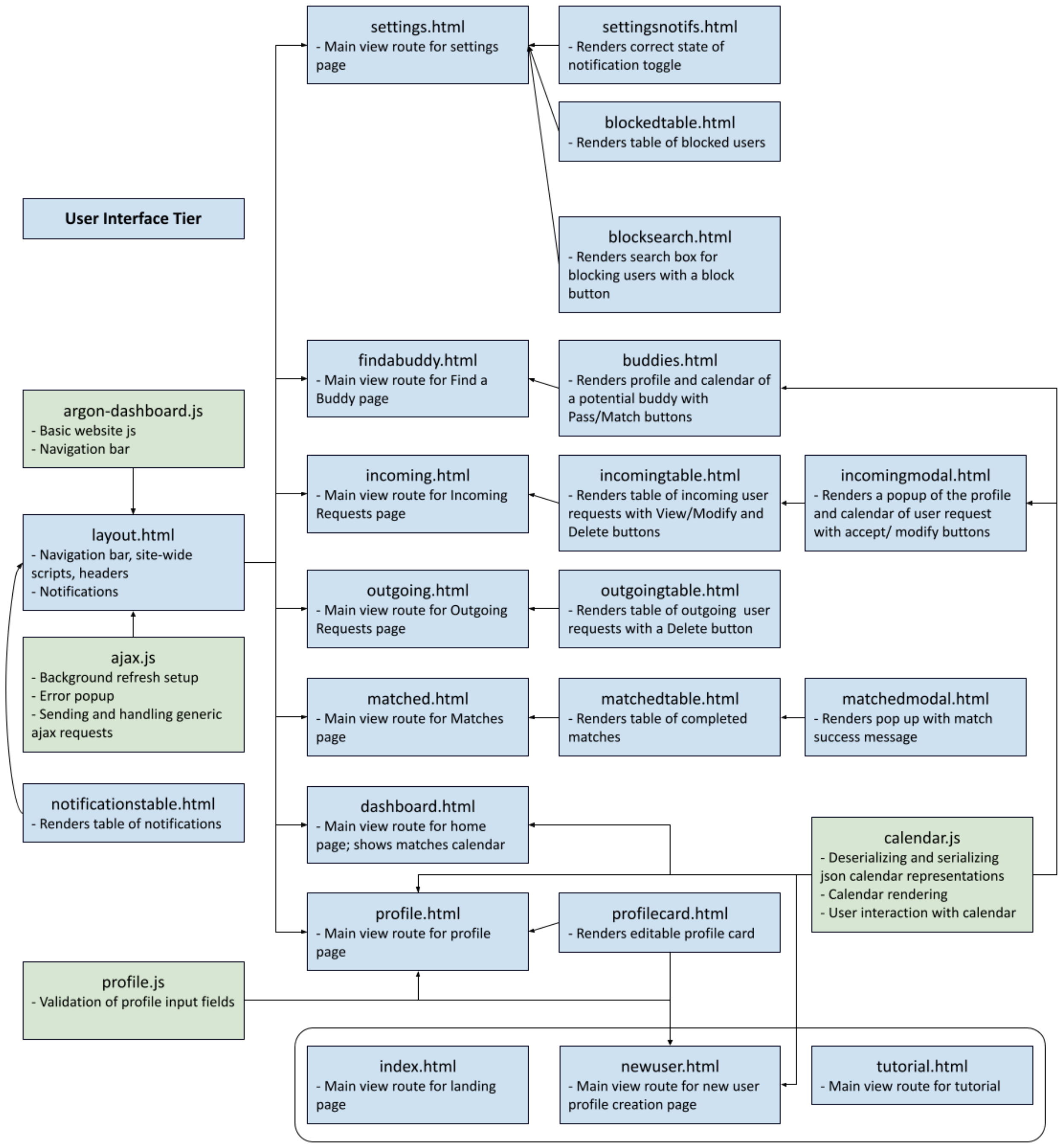
settings.html
- Main view route for settings page
settingsnotifs.html
- Renders correct state of notification toggle
blockedtable.html
- Renders table of blocked users
User Interface Tier
blocksearch.html
- Renders search box for blocking users with a block button
findabuddy.html
- Main view route for Find a Buddy page
buddies.html
- Renders profile and calendar of a potential buddy with Pass/Match buttons
argon-dashboard.js
- Basic website js
- Navigation bar
incoming.html
- Main view route for Incoming Requests page
incomingtable.html
- Renders table of incoming user requests with View/Modify and Delete buttons
incomingmodal.html
- Renders a popup of the profile and calendar of user request with accept/ modify buttons
layout.html
- Navigation bar, site-wide scripts, headers
- Notifications
outgoing.html
- Main view route for Outgoing Requests page
outgoingtable.html
- Renders table of outgoing user requests with a Delete button
ajax.js
- Background refresh setup
- Error popup
- Sending and handling generic ajax requests
matched.html
- Main view route for Matches page
matchedtable.html
- Renders table of completed matches
matchedmodal.html
- Renders pop up with match success message
notificationstable.html
- Renders table of notifications
dashboard.html
- Main view route for home page; shows matches calendar
calendar.js
- Deserializing and serializing json calendar representations
- Calendar rendering
- User interaction with calendar
profile.html
- Main view route for profile page
profilecard.html
- Renders editable profile card
profile.js
- Validation of profile input fields
index.html
- Main view route for landing page
newuser.html
- Main view route for new user profile creation page
tutorial.html
- Main view route for tutorial

Figure. User interface tier template and Javascript structure.

The HTML templates of our user interface tier almost exactly mirror our routes, in name and in terms of the relation between main view templates (e.g. incoming.html) and partial view templates (e.g. incomingtable.html), as shown in the figure above.

Overview

- **layout.html** is the skeleton for all of the pages in the gymbuddies application. It uses the argon-dashboard frontend template, as accessed through **argon-dashboard.js** in order to enforce a common UI theme throughout all the pages in the application
- **index.html** is the first page that users will be routed to when arriving at the Gymbuddies website. It serves and the signup and login page This page redirects to Princeton CAS authentication before entering the main pages of the Gymbuddies application
- **newuser.html** and **tutorial.html** serve as the pages that appear right after new user creation. It ensures that users provide information that is critical to the proper functionality of the application before entering the main pages and provide basic information regarding how to use the application
- **dashboard.html** is the "home" page for a user that has logged into the application. It displays a profile card with basic information regarding the user and has a non editable calendar that displays the current match times of the user
- **profile.html** and **settings.html** are user pages used to edit and insert their personal information and various preferences such as parameters that are used during find a buddy selection, blocking users, and enabling notifications, to name a few. **profilecard.html**, **settingsnotif.html**, **blockedtable.html**, and **blocksearch.html** are various modular components of these two pages
- **incoming.html**, **outgoing.html**, and **matched.html** serve as pages that display incoming match requests, outgoing match requests, and current matches. Importantly, associated with these pages are their associated table html pages: **incomingtable.html**, **outgoingtable.html**, and **matchedtable.html**. Their function is to allow modular refreshing of only these pages through ajax. For instance, the user may interact with the incoming table by deleting an incoming request through the delete button on the table. Then **incomingtable.html** will be sent to allow a partial refresh of **incoming.html**. Finally, **incomingmodal.html** and **matchedmodal.html** display additional popups that contain confirmations and sub operations for users to interact with matches
- **findabuddy.html** is the page where the user will enter in order to find possible candidates to send their match requests to. Like the previous pages, findabuddy.html does a partial refresh whenever the user passes on a profile card that is currently displayed on the screen. **buddies.html** is returned in order to allow the page to be partially refreshed and the current user on the screen to the replaced with the next user returned by the matchmaker algorithm
- The calendars in the Gymbuddies application are fully interactive as a result of **calendar.js**. Alongside with rendering the actual calendar as well as including the code that allowing click and drag selection along with click deselection, it implements the

transformation of data received from the calendar UI (stored as JSON) into the data format which we had designed our database in mind with
- Html pages use **ajax.js** as a gateway to make partial refreshes. Additionally, ajax.js defines preconfigured functions that when activated displays a custom error modal. Finally, ajax.js implements the crucial feature of continuous background refreshing

Ajax and Page Refreshing

In all of our pages with a logged-in user, live refreshing via a polling mechanism is provided by ajax.js. The "setup_refresh" function from ajax.js registers a list of urls to refresh, and then sends out a refresh request to the server every two seconds. A "lastrefreshed" timestamp of the last time that the browser refreshed is attached to each refresh request. The server compares the "lastrefreshed" timestamp to the "lastupdated" timestamp of the logged-in user: if the lastupdated timestamp is newer, then the server processes the request by querying the database for the necessary information and replying with the rendered partial view; otherwise, the server replies with an empty string. When the client receives the newly refreshed partial view, then they update their html and their "lastrefreshed" timestamp; otherwise, they do nothing. Only one database query is necessary for the server to determine whether or not it should update.

<u>Part 5: Design Problems Encountered</u>

1. Efficient live updating

When we initially implemented live updating, our first strategy was to simply have a polling function that performed a partial refresh every second. The issue with this strategy was that computing the information that the client requests from the server can be expensive—to the point that having several tabs open with live updating can cause the server and database to crash. To prevent this from happening, we implemented a lazy-refresh mechanism to only update the page when the user's state in the database has changed.

2. Error Propagation

When we first wrote our session decorator function, our error checking strategy was to catch any errors immediately, and then return some sort of failure value and then move on. The issue with this strategy, however, is that it requires a check to be performed after each database API call, and can easily lead to strange errors if errors are not handled accordingly. Moreover, it was difficult to determine exactly what the cause of an error was from the Flask side if, for instance, we just returned a nil value when the database API call failed. To avoid error checking after each API call and to provide the Flask application with more informative errors, we ultimately decided to propagate specific errors in each database API call to the Flask application for handling. The use of the application error handlers of Flask allow us to catch errors in all of our routes without excessive amounts of boilerplate code.

3. Database Serializability

As we discussed earlier, we chose to use the SERIALIZABLE isolation level for our database in all of our transactions. Indeed, many of our transactions are not safe against phantom reads, so strong guarantees are necessary. Because we don't maintain sequence numbers or version counters for optimistic concurrency control, this was the most straightforward way to improve the guarantees of our consistency model. On the other hand, the SERIALIZABLE isolation level can cause deadlocks to occur when multiple users try to access the same resources. Given the nature of our application, however, we reasoned that deadlocks would be unlikely, and repeated deadlocks doubly so; thus we simply give transactions a couple of chances to succeed in the case that they were interrupted by deadlock. To increase the chance that a transaction will avoid deadlock on further retries, we set a randomized, doubling delay after each failed transaction.

4. Request user flow

Our initial model for the user flow of requests was that the source user would send out a list of proposed times, and the receiving user would then choose particular times out of those proposed times to reply with. The source user would then review the proposed times and then confirm, finalizing the match. This strategy, however, involves multiple back-and-forths, even in the best-case scenario, where the destination user is already happy with the proposed times. To simplify this process, we decided to allow the destination user to confirm immediately, but we also gave them the option to propose different times. To keep things simple—for both the backend and the user—we treat the modified request almost identical to a new request, as if it had replaced the old request, just in the opposite direction. This greatly simplified the request logic and the states that we had to keep track of for requests.

5. Calendar UI

Deciding on a calendar UI was a very difficult and arduous process. We needed a calendar UI that had a drag functionality that enabled users to quickly select times, but also one that was simple enough for us to modify. Initially, we started with the Toast Calendar UI because the design itself was very clean and aesthetically pleasing. Above all else, it had a drag functionality allowing specific selection of times (i.e. 30 minute intervals). However, we quickly found that it was difficult to modify the calendar - it was built to show one-time events from week to week rather than recurring events and there was no way to make certain parts of the calendar read-only. In addition, the JS file had too many functionalities that were not needed for the scope of our application. Thus, we found a simpler implementation with a few basic functions. The final UI[2] was easy to modify such that only certain time blocks were interactable, and we found that the new implementation made the conversion process between the calendar and schedule representation in the database much simpler.

6. History

The app only keeps track of the match history. Initially, we thought of keeping a history of everything that occurred in the app. However, we found that it was not beneficial to keep track of incoming and outgoing requests. For the Incoming Requests page, a user has 3 options - accept, modify, or delete. Accepted requests would already be logged on the Matches page and modified requests on the outgoing page. Thus, the only type of incoming request to log would be the deleted requests, but this information would be of little value to the user, who likely deleted the request because they were not interested in matching with the user. For the Outgoing Requests page, we similarly did not log accepted outgoing requests, which would be tracked in Matches. We also felt that a user did not necessarily need to be informed that their outgoing request was rejected by another user. For the matches page, we decided that it would be beneficial for users to see their past matches in case they wanted to reach out to be partners again.

7. Dashboard

When we were initially discussing key pages of our application, the main landing page consisted of three large buttons leading to Find a Buddy, Requests, and Matches. However, we found that this was uninformative, as students had to go through an unnecessary extra tier to see information. After adding a side navigation bar, we found that a dashboard page was necessary rather than landing on one of the existing pages. We concluded that it would be most helpful if users could see their finalized matches at a quick glance when they first logged in, as well as their own profile shown to others on the Find a Buddy page. Seeing their own profile card could serve as a reminder to update their profile to reflect their current interests or basic information.

Sources

[1]Argon dashboard 2 by Creative Tim: https://www.creative-tim.com/product/argon-dashboard
[2]Day Schedule Selector by Artsy:
https://www.jqueryscript.net/time-clock/Create-A-Basic-Weekly-Schedule-with-Hour-Selector-Using-jQuery.html