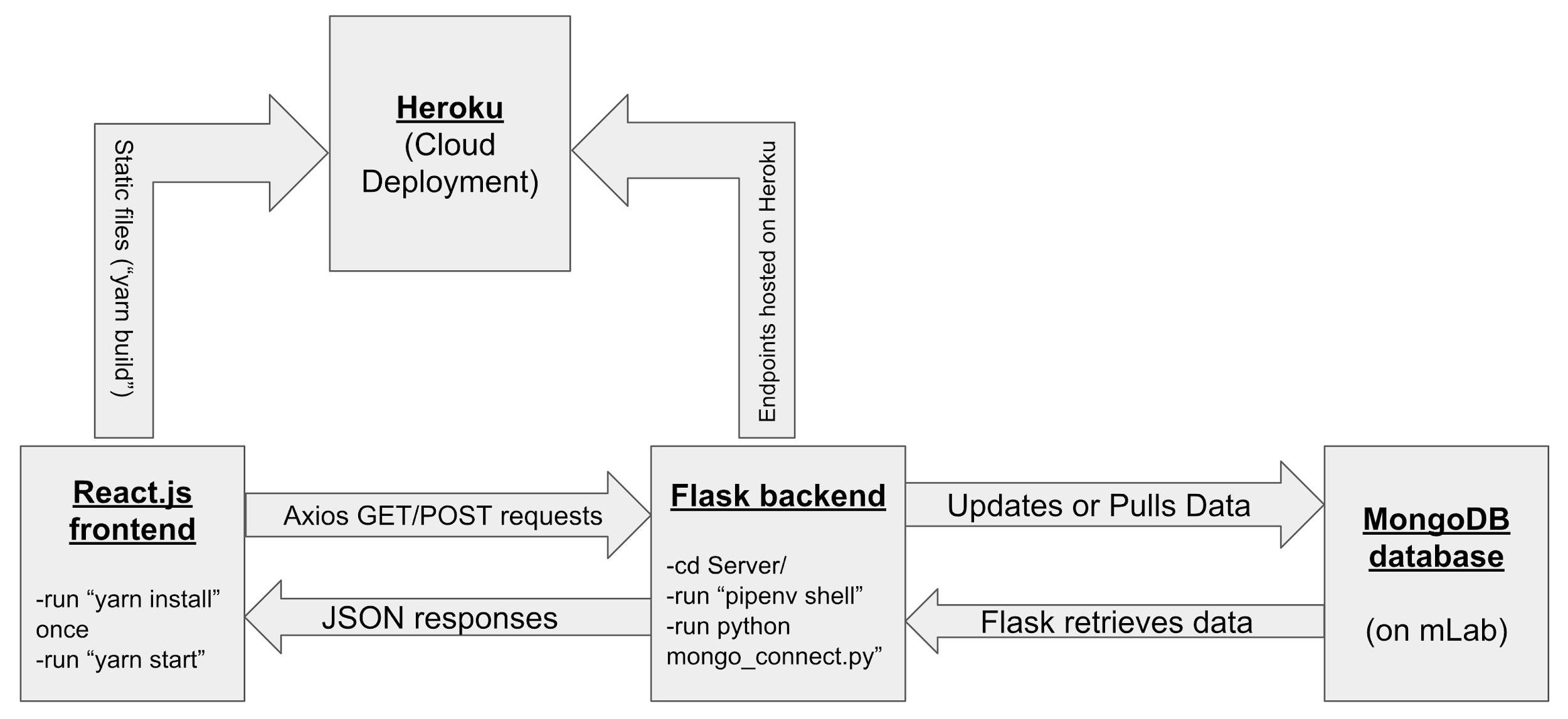
**TigerEats Programmer’s Guide**

Ishan Sinha, Paulo Frazao, Gabriel Birman, Jamison Mercurio

**An Overview of the System**

We use React.js for our application’s frontend and Flask in the backend. We also use Axios, a Promise-based HTTP client, for the frontend to communicate with the backend endpoints. Our Flask backend queries a MongoDB database, hosted on mLab for the purposes of this project. We are deployed on Heroku.



**Execution**

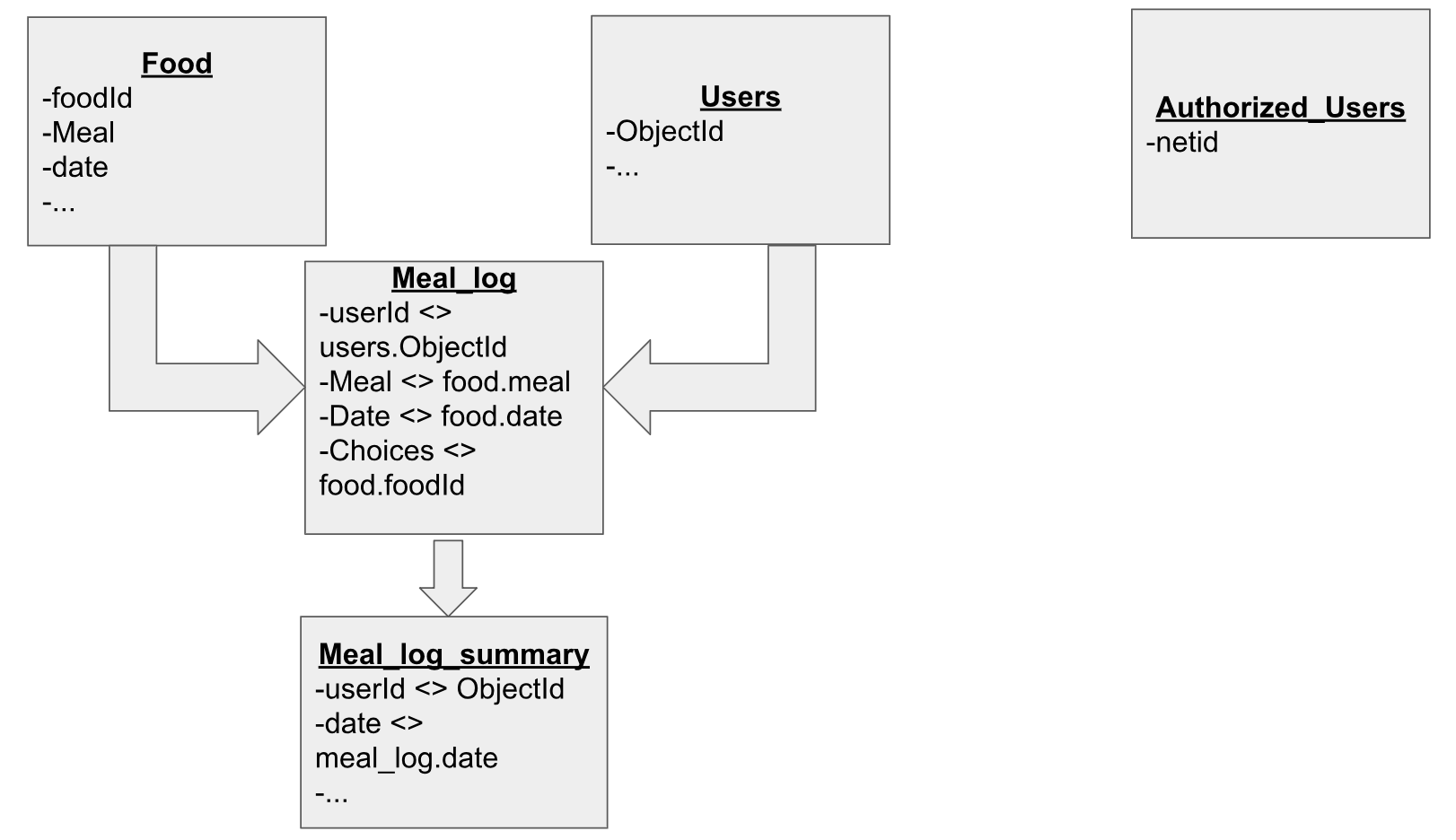
In order to execute the backend with the frontend on localhost, follow these steps:

1. Run ‘yarn build’ in the project parent directory, ‘tigereatscos333/’ to create a static bundle of the frontend
2. Cd Server/
3. Run ‘pipenv shell’
4. Run ‘python mongo\_connect.py’
5. Visit localhost:5000

Details on the inner workings of these commands are in the Frontend and Backend sections.

**MongoDB Database**

Our database is hosted on <https://mlab.com/databases/tiger_eats_db> (username: pfrazao password: y7gnykTXHj8j7EK). An internet connection is required to access it. There are 5 collections (also known as tables in SQL) in our database. We chose MongoDB, a NoSQL database, instead of a SQL database, so that we did not need to decide a rigid set of fields for each table from the start. This way, we could update data and add more fields through the development process, depending on which features we were able to get done.

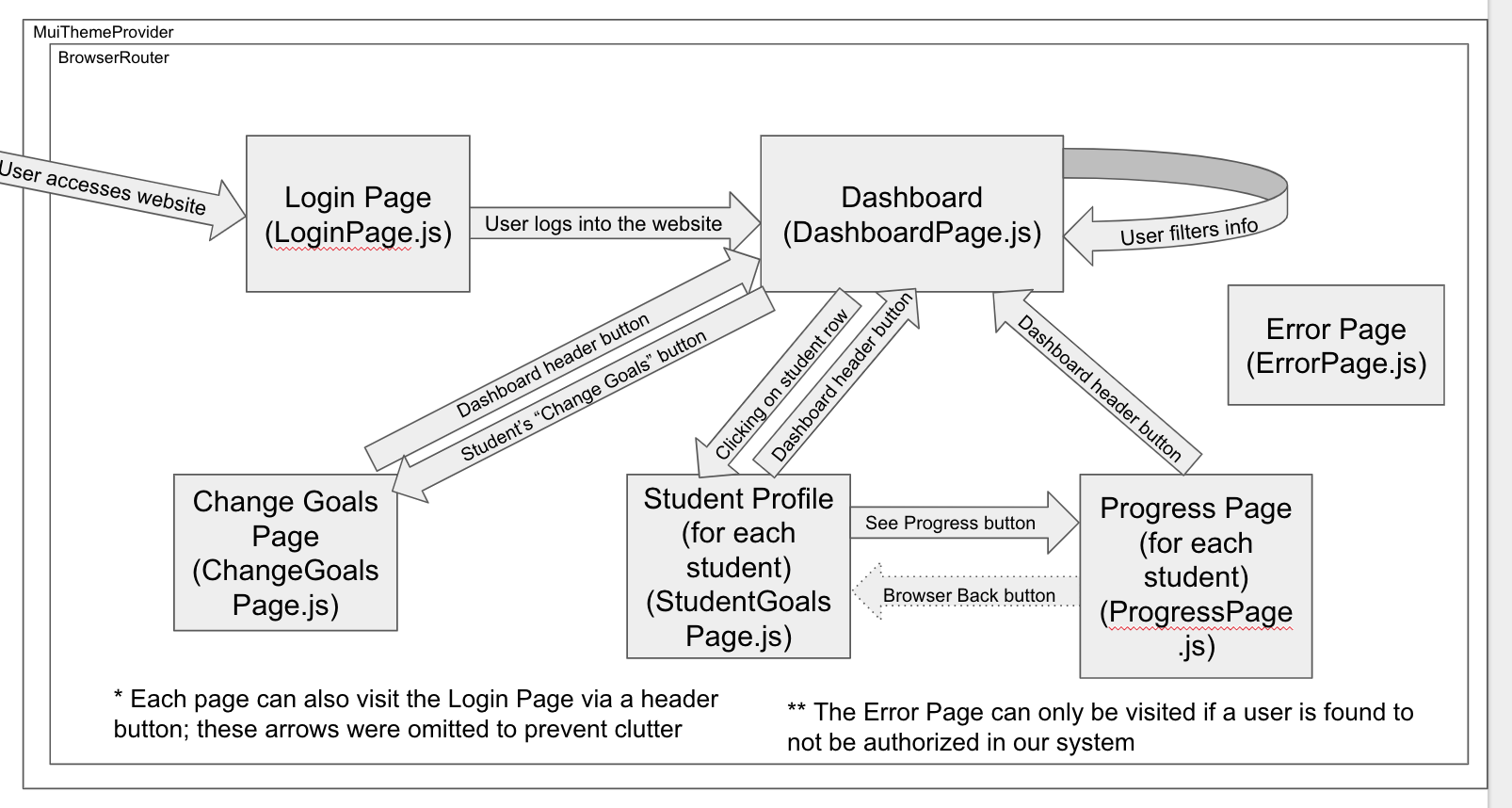


1. Users: Contains all student users of the app, their physical attributes, and their contact information. They are identified with a unique ObjectId, which is the primary key.
2. Authenticate\_users: Contains the netid’s of all authorized users of the web app, which would be Princeton nutritionists and coaches. COS333 instructors have been granted access for this project. This table is accessed after a user authenticates with CAS; the backend gets the user’s netid from the CAS session and cross-checks it with this database. This is separate from the users table because this is administrative users while the Users table is student users.
3. Food: Contains different foods’ nutrition information per serving size, along with the date and dining hall they were served at. Foods are matched based on their foodId and cafeteriaId.
4. Meal\_log: Contains logs of meals entered. Specifically, it contains a “userId” for whose meal that log corresponds to (this maps to ObjectId in Users table), a “meal” field to denote which meal it is, a “date” field to denote which day the meal was logged, and a “choices” field which is a dictionary of different food items chosen and the number of servings taken. The “choices” dictionary is queried after matching the “userId,” “meal,” and “date” fields. Then, each item in the “choices” dictionary is mapped to a food in the Food table by matching “foodId” and the “date.”
5. Meal\_day\_summary: Contains logs of daily aggregate statistics of a user’s consumption. It contains a userId uniquely identifying the user, a date, and the cumulative logged macronutrients for that day. This would usually be automatically populated in a production system, but for this project, we wrote a Python function to calculate these aggregate statistics and then populated the database with Flask. It is matched with a userId (which maps the users table ObjectId) and date.

Note: We have filled the database with mock data through 2/1/19 since we were unsure when instructors would look at our app. That is why dates may extend beyond the date you are examining this.

**Frontend** (tigereatscos333/src/)

Our frontend was created using React.js. Due to the hyper-modular nature of this library (which led to our frontend having many tiers of components, each one rendered as part of a higher parent component), the diagram for this section (pictured below) will only describe entire pages as well as communication between them. Explanations for the diagram, as well as further details about the components and flow of the frontend, are available later in this section.



While the above diagram should be mostly straightforward, a few points need clarification. Firstly, as noted in the asterisked section, a user can log out from any page besides the login page using the navigation header. Furthermore, a user may update the data displayed on the dashboard from that page itself, hence the returning arrow. Additionally, while this system makes references to the “src/styles” folder, it only does so to obtain the values of color macros and only on rare occasion; as such, it was not included in the diagram. Lastly, the entire system is wrapped in an MuiThemeProvider, which allows us to disseminate a common theme amongst all of the components, and a BrowserRouter, which handles the client-side routing (and which is further discussed below).

General

Before describing the individual modules shown above, we begin with the frontend as a whole. The frontend was coded entirely in React.js, using Axios to call relevant backend endpoints. Most of the styling in the frontend is handled within the individual modules; the application uses the “className” property to ascribe styles to elements within each component. A MuiThemeProvider component is wrapped around the entire application, however, in order to pass common theme elements down to child components (in this case, every component in the application). Similarly, a BrowserRouter component is also wrapped around the rest of the app. Our frontend uses React Router to enable client-side routing; instead of pushing page requests to the server, React Router uses a kind of conditional rendering to link buttons to pages and render them when the URL extension for that specific page is triggered. This prevents client-side refreshes when navigating through the application, and is enabled through this BrowserRouter object contained in “src/routers/AppRouter.js.” Lastly, the vast majority of the components utilized in the frontend are from the Material UI package. This library contains many pre-constructed components with many options for customization; we made use of these components to avoid constructing everything from scratch.

Execution

Our frontend can be executed locally in the following way:

1. Run the command “yarn build” in the project parent directory
   1. This method gets static bundles for the frontend

When this command is called, static bundles encompassing the frontend are returned. The backend (launched externally) serves the frontend in this static format, serving up the “public” folder in the same directory as “src.” The index.html file therein acts as a template, into which a script tag referencing index.js (in the “src” folder) is inserted. From there, the rest of the application is rendered accordingly. index.js first renders the AppRouter.js file, which handles all of the client-side routing, and all of its children are rendered conditionally afterwards (depending on the specific URL variant accessed by the user).

Login Page

The Login Page module is the page to which users are taken when they enter into the application. This page a welcome message and a simple login button, which links to CAS. Pressing this button takes the user through CAS authentication and then on to the dashboard.

Error Screen

In order to access TigerEats, the user must have additional authorization (which we track in our database) in addition to CAS authorization. If the user reaches the dashboard and is found not to have this clearance, they are bounced to an error screen that explains the problem. The user is then presented with a button to return to the login page, where they can (perhaps) enter at another point when they have clearance.

Dashboard

The dashboard, the central hub of the application, is a bit more involved. It contains a navigation header (present on all pages minus the login page), a set of filtering options, and a table containing the students with which the nutritionist is working as well as their information. Each row contains a student’s name, gender, class year, team, nutrition goals, an indicator of whether they are on the nutritionist’s watchlist, and two buttons: one to change their goals and another to send them an email. These are obtained using the “getUsers” backend function; it is called when the page first renders and its results are used to conditionally render a row for each student. The nutritionist can specify which students they would like to see by filtering on gender, team, and class year, searching for specific students using the table’s search function, or choosing to only see those individuals on the watchlist. All of these options function by tacking on “restriction” parameters to the call to getUsers. These restrictions allow the backend to narrow the list of users that it returns accordingly. They user can also sort this table using any of these fields (except the watchlist indicator and buttons); these rely on sorting logic that mimics existing implementations demonstrated on the Material UI docs.

Change Goals

When the user presses a student’s “Change Goals” button, they are taken to the Change Goals Page shown above. This page contains two options for the user to change the goals of a student: providing carbs, fats, and protein quantities, or inputting a daily caloric goal and percentage breakdowns of these same macronutrients. The first of these is more simple under the hood. After taking in these three values, the frontend inputs them into a formula and calculates the student’s new caloric goal from it; this value is recalculated every time a value changes and, if the three macronutrient fields are empty, uses their previous values. The second option is more involved. Here, the frontend performs similar calculations (albeit with different formulae) to reactively calculate and display all of the user’s new goals. When the user submits these values, the data is sent to the “change\_nutrition\_goals” backend function, where it is validated. If the data is invalid, the backend communicates a specific error message to the frontend, which alerts it to the user. Otherwise, the user is pushed back to the dashboard with updated values for that student.

Student Profile

The student profile page exists to help nutritionists see the individual details for a given student, including relevant profile info and the most recent meals that the student ate, and helps them make recommendations when they’re ready to set goals for the student, or ready to contact the student. Most of the front end here is implemented using the Material-UI React library, alike the dashboard page. Elements they have such as grids, buttons, and responsive text fields proved to be very useful in making it an intuitive layout. Calls to the backend are in axios, as mentioned earlier in this document. There are MaterialUI buttons that allow the nutritionist to email the student with a personalized message, add the student to the nutritionist’s watch list, or navigate to the progress page or change goals page for this particular student. On the page, we display general profile information about the student stored in our database (for instance, the student’s team, age, weight, goal weight, and goal macronutrients). In addition, I used a horizontal bar graph through the React-Vis library to display the student’s progress on reaching those macronutrient goals (we had trouble rendering this on heroku, so it’s been changed to a more basic horizontal bar graph). Moreover, this page provides the nutritionist with a breakdown of the last five logged day’s meal data. This includes

* The day’s macronutrient intake breakdown up at the top
* The macronutrient breakdown for every meal
* The foods eaten and their corresponding serving sizes
* A picture of the meal that the student took themselves (we don’t have the data for that yet, so it’s currently a placeholder image of some bread and eggs).

Progress Page  
From the student profile page, the user can select to see the progress page, which relates the student-specific nutrient intake as a time series with some useful statistics, which helps the nutritionist contextualize the student’s eating behaviors and extract a greater variety of meaningful data. Upon calling the ProgressPage component, the component will query the server for the student’s entire time series data and for the student’s nutrient goals. While querying this information, the component will render the navigation header and a loading circle. After the information has loaded, the screen will render and display the following sub-components: the time series chart, the brush range chart, and a menu. The time series data is aggregated using a library called pond.js, which thereafter feeds into the complementary time series library React TimeSeries Charts to display the data. Both of these are open-source libraries provided by ESNet. The screen is responsive to width changes, but maintains a fixed height.

The initial view of the times series chart has four data channels color-coded red for protein, yellow for fat, light green for carbs, and dark green for calories. These displays the entire time range available for the given student as bar charts overlaid with a dotted line representing the student’s goal amount for that particular nutrient. Each data channel contains a label axis on the left, and a value axis on the right. The label axis itself is divided into two sections. The first section displays the min and max values of the chart on the Y-Axis, where the max is the greater of the max value in the series and the goal amount. The second section, to the left of the Y-Axis, is a set of statistics such as the minimum value, maximum value, and average value of the displayed time range, and lastly the goal amount. The value axis displays the value (with appropriate units) for the given bar specified by the tracker line, which follows the time index referred to by the mouse pointer during hover events over the time series. Thus, the tracker line synchronously updates the values for each data channel, and outputs the given time index as a date string in the upper menu bar, with the string being dependent on the whether the rollup (see below) is daily, weekly, or monthly. Note that the label axis statistics represent an average over the entire displayed time range in days, whereas the value axis represents the average for the given rollup.

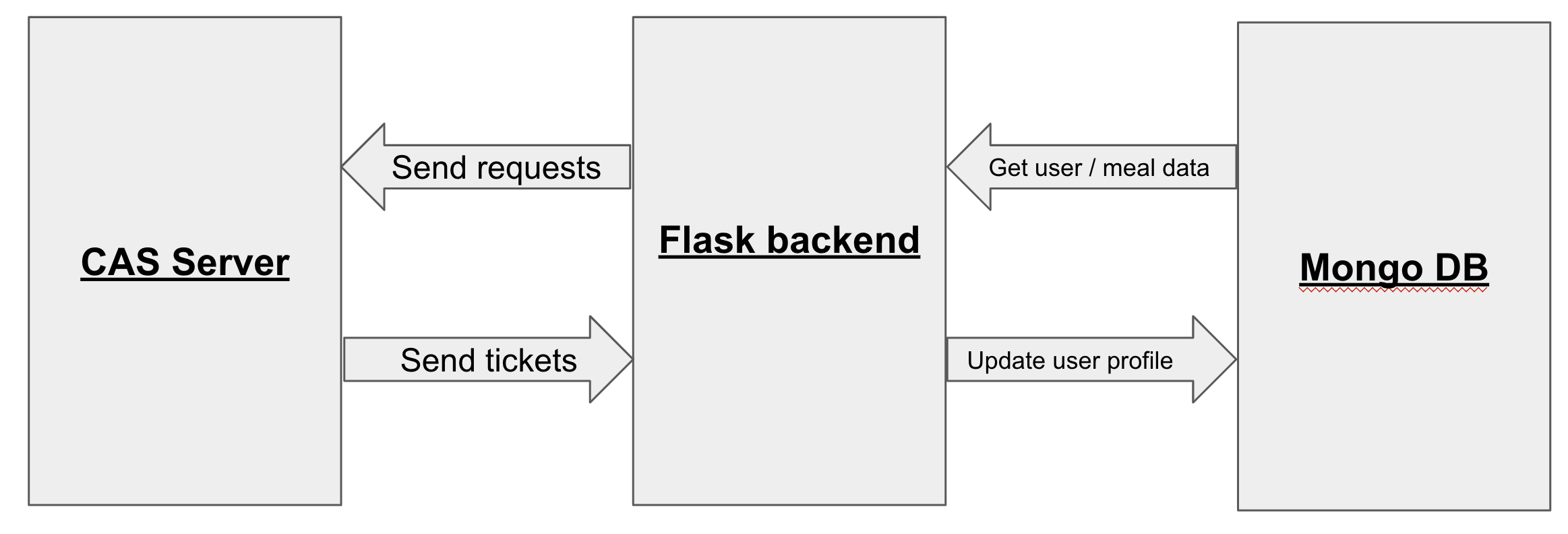
The user is able to update the time range displayed for all charts simultaneously by scroll-zooming on the chart container, or panning the brush displayed below the chart (this feature is not available on small screens, however). The minimum resolution for the zoom is one day. The brush is overlaid onto a bar chart which displays the entire time series of stacked nutrients using a daily rollup (see below). A clickable help modal exists to explain this setup to the user.

On top of the time series is a menu bar which contains three sub-components: the rollup time window radio buttons, the toggle channels checkboxes, and the tracker time label.

The rollup time window radio buttons allow the user to “rollup” the time series into either daily, weekly, or monthly averages, with the default set to daily. The daily rollup displays the average values for each day. The weekly rollup currently displays averages of each week starting on a Thursday and ending on the subsequent Wednesday. The monthly rollups provide an average for each calendar month. Selecting one of these radio buttons causes the widths of the time series bars to resize accordingly, and also changes the date format of the tracker time label to an appropriate format.

The toggle channels checkboxes are 4 checkboxes color-coded with the color of the corresponding nutrient. Though they are all set to “display” by default, where a check mark indicates that the nutrient should be displayed, the user has the option to remove the chart from the display, upon which the remaining channels will be vertically resized to fill the container height. If there is only one visible channel remaining, the check box is locked to ensure that there is always a minimum of one nutrient being displayed.

**Backend** (tigereatscos333/Server/)



Setup and execution

Our backend was built on Flask in a file called “mongo\_connect.py”. It receives GET and POST requests from the frontend via Axios, and returns JSON responses to the client. The package requirements for the backend are contained in Pipfile and Pipfile.lock. In order to run the backend locally, one must change directories in a terminal to ‘tigereatscos333/Server’, run ‘pipenv shell’, and then run ‘python mongo\_connect.py’. The backend runs on port 5000 locally. In the declaration of the app in the backend, we link the Server to a static build of the frontend (see the above Frontend section) so that both run on port 5000. Thus, one should visit localhost:5000 to run the application.

Database

Interaction with the database is handled with PyMongo. This takes our login information for mLab on MongoDB and allows us to query our database. Queries follow PyMongo syntax.

CAS

We implemented CAS by adjusting CASClient.py from lectures to work with React and Flask; we import this in mongo\_connect.py. We adjusted the authenticate() function in CASClient.py so that it returns None if the user is not in an authenticated session with CAS as opposed to redirecting them to a CAS login with a ticket. We did this so we could create a decorator in CASClient.py that a) redirects a user to CAS login if he is not logged in and then runs the decorated function after the user logs in; or b) runs the decorated function if the user is in an authenticated CAS session already. Specifically, we created Flask route ‘/<path:path>’, that catches any URL that begins with ‘localhost:5000’ when running locally and ‘<https://tigereats.herokuapp.com/>’ when on Heroku. We also created ‘api/login\_casclient’, which directs the user from the Login page to the Dashboard after clicking the “Login with CAS” button. We decorated both of these routes with our decorator so that accessing any TigerEats affiliated URL, whether it is a legitimate page of the app or not, requires CAS authentication, thereby fully protecting our application. We had to create a global in instance of the CASClient class, called casClient, in mongo\_connect.py so we could decorate functions in mongo\_connect.py.

However, authenticating with CAS is only the first step. Not everyone in the Princeton community should have access to this web application -- only select nutritionist and coaches should. For this project, we are giving 333 instructors access as well. We made a list of netid’s of authorized users in our authorized\_users table in our MongoDB database. After a user authenticates with CAS, we get their netid from the session. The frontend then calls the backend endpoint, ‘api/user\_role’, which determines if that netid is in the list of authorized users’ netid’s. If it is, the frontend renders the page. If it is not, the frontend sends the user to an error page that tells the user he/she is not allowed to view the web app.

Sessions

To implement sessions, we used ‘beaker’ and ‘flask.sessions’ to create a sessions interface, BeakerSessionInterface, for the app. The session\_options are the same as in the example penny.py. Ultimately, we implemented sessions in the exact same way as was implemented in the PennyReact example, just with Flask instead of Bottle with beaker.

Groups of major functions

We have a few major sets of Flask-routed functions:

* CAS login functions. These are used for session-based authentication (‘<path:path>’, ‘api/login\_casclient’, and ‘api/user\_role’)
* User data retrieval methods (GET): These are used to give data about individual users and sets of users (‘/api/getUsers’, ‘api/get\_all\_user\_info’)
* Meal log data retrieval functions (GET): These are used to return data regarding meals (‘/api/get\_user\_day\_meal\_data’, ‘api/get\_user\_nutrient\_progress\_all’)
* Update nutrition goals function (POST): This is used to update a specific user’s nutrition goals (‘/api/change\_nutrition\_goals’)
* Update student watchlist status function (POST): This is used to add or update a note a student’s status on the watchlist (‘api/change\_watchlist’)

We also have non-Flask routed functions that we used to populate the database:

* Database population functions: These are used to fill the database with realistic mock data (\_fill\_database\_meals, \_fiill\_database\_daily\_summary, \_delete\_items).

**Environment and Package Management**

Our React dependencies are maintained using yarn as a package manager, which makes use of the package.json and yarn.lock files to construct a reproducible node\_modules tree. The package.json file stores meta data, top-level dependencies, and various configuration information. The yarn.lock file stores dependencies of dependencies in order to “lock” the tree in the face of updates.

Since our server is written in the Python language (with yarn being standard for javascript usage), we use the pipenv tool to maintain a virtual environment that maintains the same python version and dependencies across users. The Pipfile and Pipfile.lock effectively occupy the same niche in the python realm as package.json and yarn.lock do for the React side.

**Heroku**

We chose heroku as our deployment platform for the project, as it has a free tier that proved useful when hosting our React front-end with our Flask back-end. We initially had plenty of configuration issues with Heroku, and online documentation mainly recommended purchasing a second web dyno in order to have our React communicate with Flask, while running on different ports. However, after communicating with another group, we found the most efficient way to host our program was on one dyno, having our react front-end hosted via a static javascript build, while Flask serves these files to the client browser as needed (we used Gunicorn as the WSGI HTTP server, rather than Flask’s built-in developmental server). This resolved some memory leak and service time-out errors we were facing initially (see “Design Problems” below for details on this)

The two Procfiles are used in order to have both the React front-end and Flask back-end deployed on the same Heroku web dyno, as described in this link: <https://medium.com/@nadayar/heroku-fu-multiple-servers-on-one-dyno-6fc68d57b373>

Put simply, we used the Python port version of Ruby’s “Foreman” server process manager, called Honcho. Our default Procfile (which Heroku sees, and executes) runs Honcho, which then runs the two processes in ProcfileHoncho (the static React build and the Flask server). This was the best fix we could find when we encountered a plethora of errors while running both React and Flask simultaneously on the same web dyno. Another, slightly simpler solution would have been to pay for a second web dyno from Heroku, but where’s the fun in that :)

**Design problems**

* **Frontend**
  + Navigation Header - I had some trouble ensuring that the header would only render on certain pages (in this case, all of them except the login and error screen). At first, I tried a lot of Reactful solutions, but most of those led to more complicated issues of state that seemed unnecessary for my purposes. So, I used the window.location.pathname key phrase available in JS (which returns the path of the page currently loaded) and set the navigation to conditionally render only if this pathname did not correspond to the error or login pages; if the user were on those pages, then, it would return null and simply not render.
  + Authentication - We wanted to ensure that only those users who were both CAS-authorized and were present in an “authorized” database could view pages; otherwise, ideally, they’d be bounced to an error page. I was having a lot of trouble with this redirect, and I ended up trying many different implementations that relied on conditional rendering. Based on a great idea from Jamie, though, I was able to avoid conditional rendering altogether using React Router and Axios. Using a React lifecycle method, each page, upon rendering, checks to see if the user is authorized and bounces them to the error screen if not, allowing them only to return to the login page. Unfortunately, this is limited by the fact that an unauthorized user will catch a very quick glimpse of the screen at the outset; however, since the data is not populated at that point, the solution seemed to work well enough
  + Filters - I worked on the dashboard filters very early on in the project and they turned out well! The issue, however, is that it took a bit for us to establish how we would be sending the filter information over to the backend. We settled on sending a dictionary of filters, however updating that particular piece of state took a bit of doing simply because working with dictionaries in that capacity is not something I was used to at the time. Thankfully, it worked out, and the filters now look pretty good and function well.
  + Progress Page
    - The time series displays always seemed to be shifted to the left by 5 hours or so. This turned out to be an issue with UTC vs. local time that was corrected by displaying everyone’s data relative to UTC time. This was greatly facilitated with the use of the moment.js library.
    - The library enabled an infinite zoom resolution with the time series, but I incorporated a minimum resolution of one day, as well as snapping the time series to the start of each day during panning so that it was never possible to use sub-daily resolutions. Moreover, I set up a minimum and maximum day for the time series to prevent users scrolling outside the acceptable range. However, this came at the cost of a “stuck” scrolling zoom out feature on when zooming in on the earliest day of the series.
    - Initially, we had configured the data such that a null entry from the user for a given day set the nutrient values to zero. However, this conflicted with the possibility that the user wanted to fast either for religious or nutritional reasons (see: Ramadan/Passover for religion, or intermittent fasting for nutrition). Thus, we decided to allow for null entries, and incorporated a filter into the aggregation tools that ignored null data in the statistics.
    - Initially, the data was presented using a stacked area chart (as seen in the beta). However, I instead chose to use 4 different bar chart channels to represent each nutrient as this would maximize visibility for the user. After all, it did not make sense to stack data with different units in the same graph, especially when levels of one nutrient would often largely outweigh levels of another, making it difficult to track progress for smaller-valued nutrients. Though the brush chart in the bottom does contain a stacked bar chart, the actual values are irrelevant as it is only the time series that is important, so this did not confuse the design.
* **Backend:** 
  + The only issues arose from the Heroku misconfiguration (explained below)
* **Heroku**
  + Static Frontend: Initially, when running this locally, we would run ‘yarn start’ in the tigereatscos333/ directory to run the frontend on port 3000, and run ‘python mongo\_connect.py’ in another terminal in tigereatscos333/Server/ to run the server on port 5000. With these running as separate processes, the backend could not redirect the frontend to frontend endpoints -- it could only redirect to backend endpoints -- so CAS was not functional because we could not redirect the user to the correct frontend page after authenticating. Instead, we’d send them to a nonexistent backend endpoint. We also had Heroku memory issues with this method since we would have to deploy the whole frontend as a process to Heroku. We solved this by replacing ‘yarn start’ with ‘yarn build’ to get a static build of the frontend when running locally; we removed ‘yarn start’ from the Procfile so that the frontend would not be run as a separate process on Heroku. We linked our static build to our backend through our app declaration in mongo\_connect.py. With this, both the frontend and backend run as one process, enabling the backend to redirect to frontend endpoints, which enabled CAS to work. This also resolved our memory issues since the static frontend consumes significantly less memory than it does as a separate process.
* **Database:** 
  + No issues for implementation of our application, although we did not precisely follow the database setup criteria we learned in class since we use a NoSQL setup.