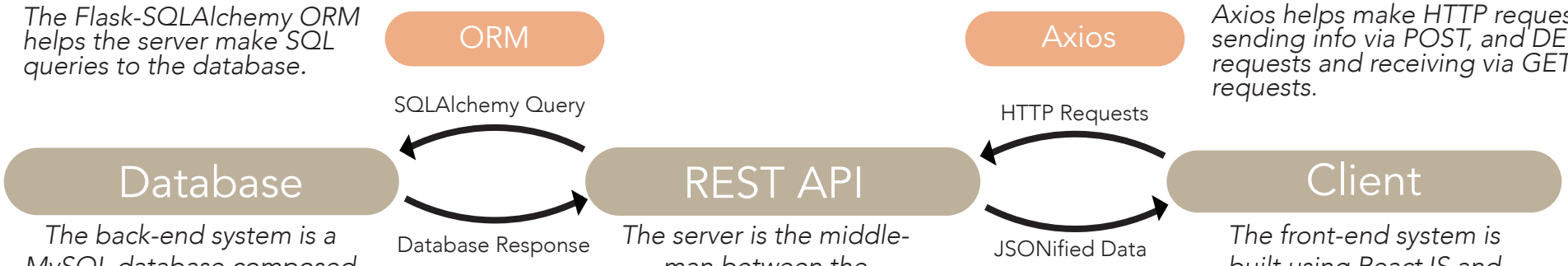
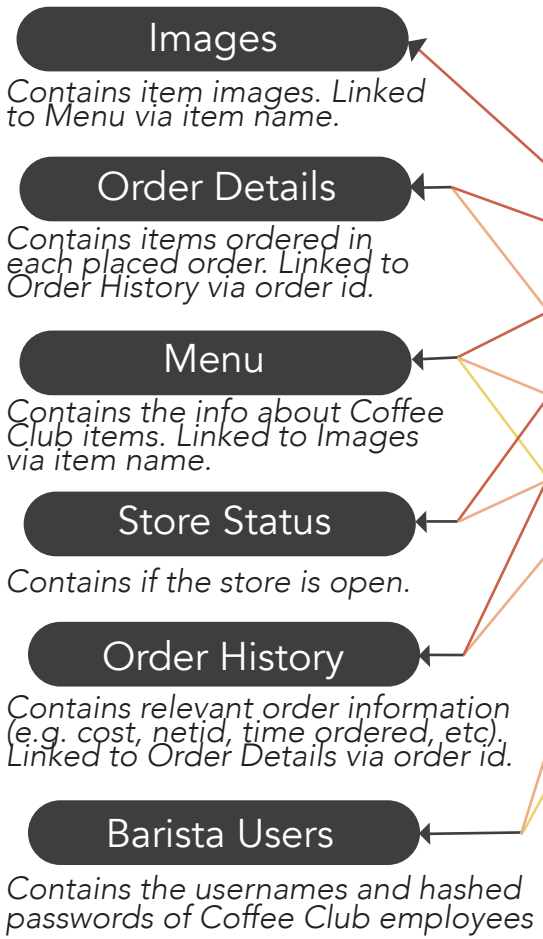
EXPRESSO

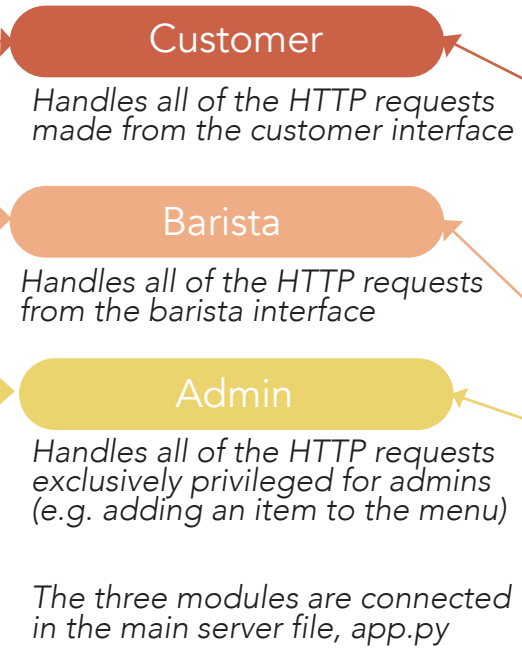
PROGRAMMER'S GUIDE

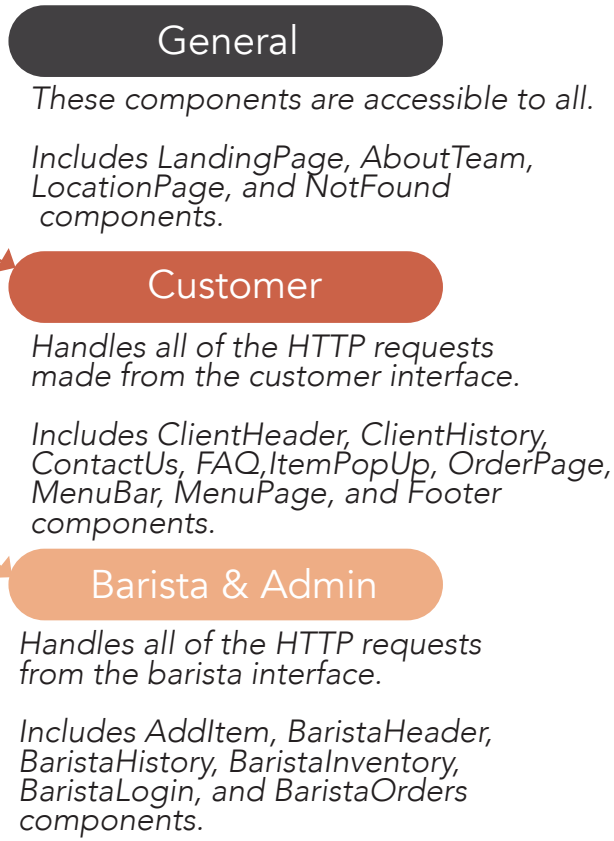
victor hua, joseph kim, hari raval, karen ying, & dora zhao

TOP LEVEL DIAGRAM









Overview

Our web application is divided into three tiers: backend system, server-side, and the frontend system.

The web application is hosted on cPanel— the University's web hosting technology. We are using cPanel rather than other hosting services,such as Heroku, because we are implementing student charge and accessing LDAP. Hosting on cPanel allows us to fully and securely integrate with the University's technologies.

As illustrated in the boxes-and-arrows diagram, the three tiers of the web application interact with each other to create Expresso. In the middle we have our server— the REST API built on top of the Flask microframework. It connects the backend and frontend systems. On the backend, we have a relational MySQL database that stores information about the menu items and placed orders. Using an object relational manager, the server queries information about the database. It then serializes the data into a JSON format and sends it over to the frontend. The frontend, located at www.coffeeclub.princeton.edu is made using React and consists of several components. The two main buckets are the customer interface and the barista interface. In turn, the frontend communicates with the server via HTTP requests. By sending the information from the database to the frontend via the server, we are able to give functionality to the Expresso web application.

Backend Systems

DATABASE (database.py, models.py)

Our database is stored on cPanel as a MySQL database. We have six tables that have the following structure:

1. The **"images"** table contains a picture of a menu item in each row; there are two columns with the image name stored as a VARCHAR (255) and the image encoded as a BLOB (binary large object) type.

2. The **"menu"** table contains all the information associated with the Coffee Club menu in seven columns. Each row has the details associated with an item on the menu. The table includes the item size, item name, category (food, beverage, or add-on), item price, availability, definition, and description. Aside from item price which is stored as a DECIMAL and item availability which is stored as a BOOLEAN, all other fields are stored as a VARCHAR.

3. The **"order_details"** table contains the information associated with each person's order. Each row contains an individual item from an order, where we store the name of each item as VARCHAR, the order id that ties a multiple item order together as an INT, and each menu item's unique id as an INT.

4. The **"order_history"** table stores whole orders in each row. We store the student's netid as a VARCHAR, the order id as an INT, the time the order was placed as DATETIME, the total cost of the order as a DECIMAL, the type of payment (student charge or in-store) as a BOOLEAN, the payment status as a BOOLEAN and the current status of the order (ready, in progress, not complete) as an INT. "order_details" and "order_history" are connected by the order id field. This allows us to see individual items in the "order_details" table that are associated with each order in "order_history".

5. The **"valid_barista_users"** table contains the usernames and encrypted passwords of barista users. Both columns stored as VARCHAR.

6. The **"store_status"** table has one BOOLEAN column indicating whether the store is open.

We update some columns in the database via POST requests . The order status column in order_history table is an INT of three possible values: 0 (not complete), 1 (in progress), 2 (ready). This value is updated when, the barista clicks a button to change the status, sending over the updated value. Similarly, the payment_status boolean in order_history can be updated via an event on the frontend. All students who pay with student charge are automatically marked as paid. However, for students who pay in store, their payment_status is initially false and is changed to true once the frontend triggers the POST request to update the database. Finally, in the menu table, availability indicates whether an item is in stock. When a barista toggles the availability button, this sends a POST request that changes an item's availability from in stock to out of stock or vice versa.

DATABASE TABLES

IMAGES

item*	Character
picture	Blob

BARISTA USERS

username*	Character
password	Character

MENU

size*	Character
item*	Character
price	Decimal
availability	Boolean
category	Character
description	Character
definition	Character

STORE STATUS

store_open*	Boolean

ORDER DETAILS

order_id*	Integer
item	Character
item_id*	Integer

ORDER HISTORY

netid*	Character
order_id	Integer
timestamp	Datetime
total_cost	Decimal
type_of_payment	Boolean
payment_status	Boolean
order_status	Integer

* = primary key

FLASK-SQLALCHEMY ORM (models.py)

We used SQLAlchemy, an Object Relational Mapper (ORM), to easily access our MySQL database within our Python script. Specifically, we used the Flask-SQLAlchemy package to integrate it into the Flask framework. In models.py, we declare our schemas, which correspond to the tables in the database and the different columns. The ORM is used throughout the server implementation to query the database, whether it be getting information from the database or adding/ deleting new information.

LDAP (ldap.py)

LDAP (Lightweight Directory Access Protocol) is a networking protocol that allows us to query and modify directory services running over TCP/IP. In order to access, integrate, and display personalized information on both the barista interface and customer interface we used Princeton's LDAP which gives us access to query information associated with members of the Princeton community. After authenticating and connecting to the University server, we retrieve the following fields from the server: university ID number (PUID), display name (nickname, if provided), last name, full first name, email address, enrollment status (undergraduate, graduate, etc.), and class year. Our use of LDAP occurs in LDAP.py in which we create an "LDAP object" that permits quick access to all of the fields mentioned above based on a queried netid.

PAYMENT INTEGRATION (payment.py)

Our payment code relies on a cron job. Cron job is a Linux utility which schedules a command or script on a server to run automatically at a specified time and date. Once a day, at 11:59 p.m., Cron job executes a script which runs our payment code, processing the payments from that day. Our script creates a ".mp file" which contains the in-store orders that have not been paid for as well as the orders that deal with student charge. The files are created and stored in a password protected director for the IT manager at ODUS to process. Our payment utilizes LDAP and the fields retrieved from the server to ensure that the person being charged is a valid student.

DAILY REPORT CSV (dailyReport.py)

We utilize Cron job to create a CSV of online daily sales. This cron job, similar to the cron job that handles the payment script, is run every 24 hours at 11:59 p.m. The script organizes all orders from the current day by order id and includes the time each order was placed, the items ordered, the dollar value of each order, and the payment type (student charge or in-store payment). Additionally, the script tabulates the total daily sales, in dollars. This CSV is sent to the Coffee Club president by email each day.

For maintenance purposes, every year, a programmer must update the global variables of the different student years to reflect the current graduation years of students enrolled (i.e. 2020, 2021, 2022, and 2023). This allows us to check whether a student is currently enrolled at Princeton. Additionally, the programmer must update the global variable of the last day seniors are able to use student charge. This is required by Princeton administration since students cannot use student charge after a set day in April.

Server Components

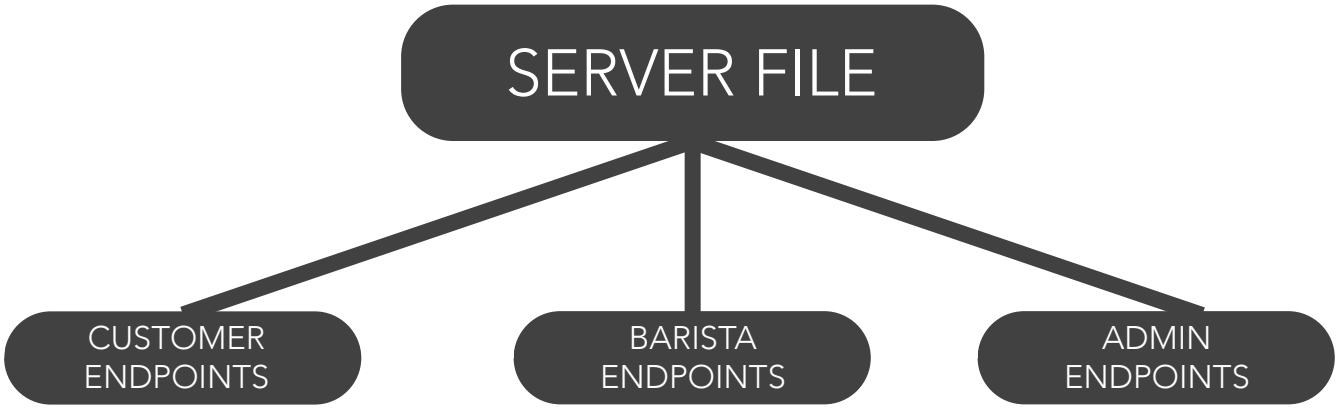


DIAGRAM: OVERVIEW OF SERVER COMPONENTS

REST API (app.py, customer.py, barista.py, admin.py)

The RESTful API is built using a Flask micro web framework in Python. The API is broken into three different modules— customer, barista, and admin— representing the three different users for our web application. As follows, all requests coming from the customer interface begin with a "/customer" prefix, barista requests begin with "/barista", and admin "/admin."

The RESTful API is accessed via HTTP requests (GET, POST, and DELETE) from the client-side and return data in a JSON format. Flask-Marshmallow is integrated with the Flask-SQLAlchemy ORM to define schemas that help serialize / deserialize the data being sent across the HTTP requests.

The table on the following page enumerates the different query strings that can be used with the REST API.

AXIOS (axios_getter.js)

We connected the frontend with the server using Axios, a Promise-based HTTP client that integrates with React. Axios makes XMLHttpRequests and allows for JSON data to be sent across the server. For GET requests (ex: /customer/menu to get the items the Coffee Club offers), we would call the query string and receive the data in a JSON format. For POST requests (ex: /barista/<order_id>/inprogress marks an order from unstarted to in progress), we sent data from the front-end in a JSON format and made changes to the database. Finally, on the admin side, we also used POST and DELETE requests to change menu offerings in the Menu table. A more detailed listing of the REST API is included on the following page.

SERVER API

CUSTOMER API ('/customer')

Name	Type	Description
/authenticate	GET	Uses CAS Client to log the user in and redirects to the menu page
/getuser	GET	Returns the netid and JWT access token for the user
/logout	GET	Log out user using CAS Client
/menu	GET	Returns available items with their respective images
/menu/<item>	GET	Returns specific Menu details about an item
/storestatus	GET	Returns a boolean telling whether the store is open
/<netid>/placeorder	POST	Checks the availability of the items in the cart before sending a POST request where *netid* is the individual placing the order.
/<netid>/orderhistory	GET	Returns all of the orders a customer has made
/<netid>/displayname	GET	Returns the display name of the customer queried using LDAP
/customer/contact	GET	Gets the contact message and deliver it in an email to the Coffee Club inbox. Returns a confirmation the email was sent.

BARISTA API ('/barista')

Name	Type	Description
/authenticate	GET	Checks whether the username and password information match any user in the database. Returns the JWT access token if successful or 'Invalid Login' message if not.
/getuser	GET	Returns the username of the user logged in.
/logout	GET	Clears the user from session
/getorders	GET	Returns all orders that are uncompleted, unpaid for, or both
/<id>/getorderstatus	GET	Returns the status of an order (0 = not started, 1 = inprogress, 2 = complete)
/<id>/complete	POST	Sends a POST request where *id* is the order to be marked as complete
/<id>/paid	POST	Sends a POST request where i*d* is the order to be marked as paid
/<id>/inprogress	POST	Sends a POST request where *id* is the order to be marked as in progress
/<item_name>/getstock	GET	Returns whether *item_name* is available
/<item_name>/changestock	POST	Sends a POST request changing the availability of *item_name*
/getdayhistory	GET	Returns all the orders placed that day
/loadinventory	GET	Returns all the items on the Coffee Club menu
/storestatus	POST	Toggles the open status of the store

ADMIN API ('/admin')

Name	Type	Description
/checkstatus	GET	Returns whether the user has admin privilege
/addinventory	POST	Sends a 'POST' request, adding inputted data as a menu item
/<item_name>/ deleteinventory	DELETE	Sends a 'DELETE' request, removing *item_name* from the inventory

UNIX EMAIL SENDER (sendmail.py)

Email notifications are implemented using sendmail. sendmail is a mail transfer agent, on Unix-based systems, that is used to route email over the Internet to a specific recipient and can be done by running the sendmail command in a Shell environment or within a programming script. As sendmail is not intended as a user interface routine, we create pre-formatted messages for each email type: order confirmation, daily sales, and feedback. To pre-format the messages, we utilize Multipurpose Internet Mail Extension, particularly the submodules MIMEMultipart, MIMEText, and MIMEBase.

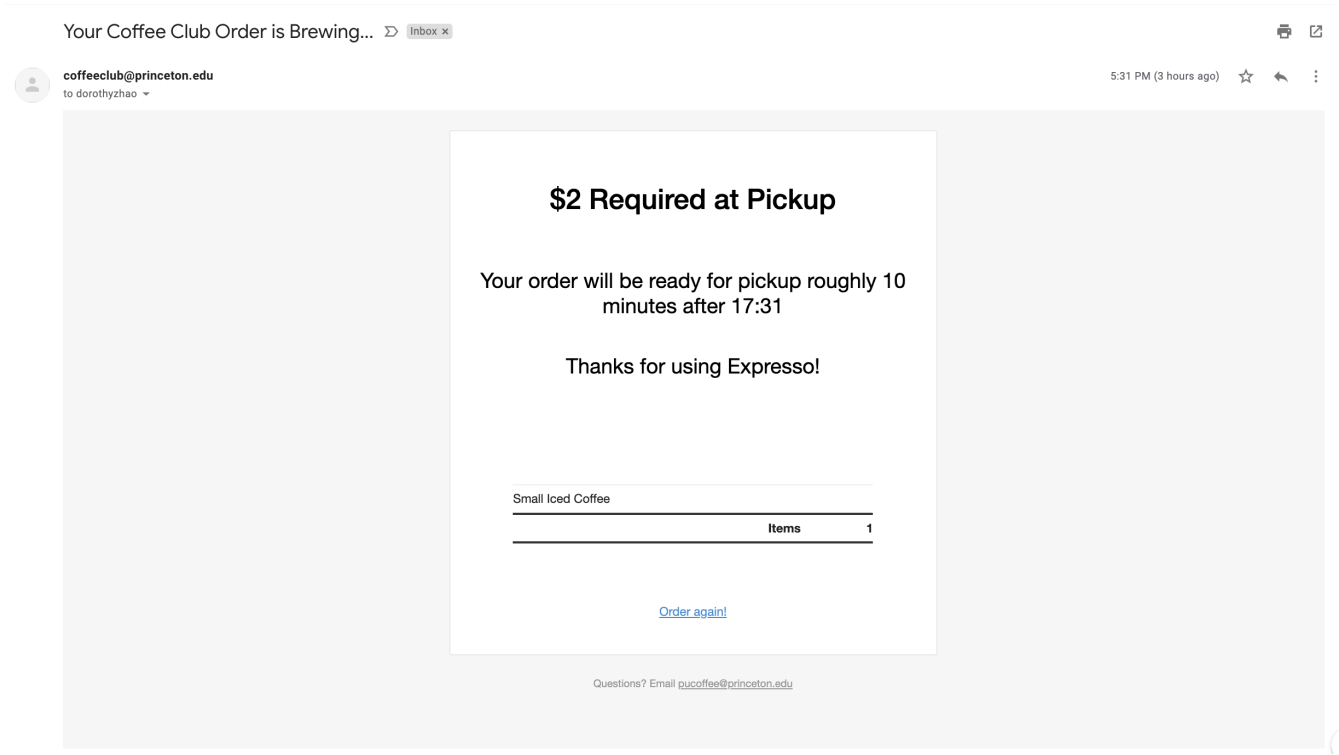


DIAGRAM: EXAMPLE EMAIL THAT IS SENT TO CONFIRM AN ORDER HAS BEEN PLACED

Frontend Systems

REACTJS & SEMANTIC UI

Because we knew this project would be front-end heavy, we chose to use ReactJS. This allowed us to frequently reuse components, such as ClientHeader or Barista-Header, across different pages. While many React apps use global state managers, like Redux, we do not because we mainly passing around two states— the shopping cart and user information. However, we do use local storage to store the customer's shopping cart during their active session.

We also used Semantic UI-React, the React integration of Semantic UI, a development framework for creating web layouts. Semantic UI provided us with predefined tools, like Grid, Button, Sidebar, and more, that we used to design the pages. All of the components that we used are Semantic components. Furthermore, Semantic allowed us to easily create mobile responsive pages since we knew people would be viewing the web application on their phones. On mobile devices, we change the layout to make the website easier to navigate on small screens.

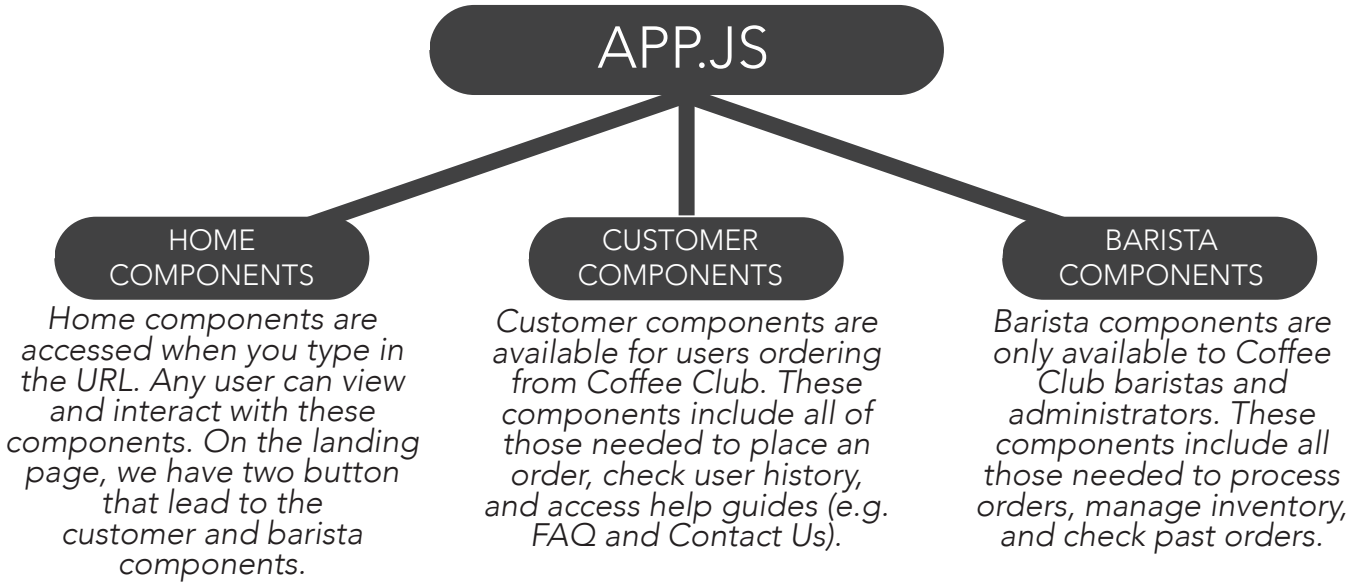


DIAGRAM: FRONTEND OVERVIEW

COMPONENTS

The components are split into three main sections: Home, Customer, and Barista. As illustrated in the diagram, there is a main App.js file that we use to render the different components for the three sections. On the following page we detail what each component does.

FRONTEND COMPONENTS

HOME COMPONENTS

Name	Description
LandingPage	Default page users are brought to.
AboutTeam	Renders the About Team page. Includes team members, acknowledgements, and a message from the developers.
LocationPage	Gives instructions to locate The Coffee Club, along with a Google Map with the location pinned.
NotFound	Default error page for any page not found on the Expresso website.

CUSTOMER COMPONENTS

Name	Description
ClientHeader	Navigation bar at the top of every page on the client side. Allows for navigation to the landing page, menu, FAQ page, contact us page, user history, cart page, feedback Google Form, and client logout.
ClientHistory	Displays every order the client has placed. If the client has not made any orders, a "No Orders!" message is displayed instead.
ItemPopUp	Renders popup on the client menu where customers can view an item, select add-ons and sizes, and add the item to their cart.
MenuBar	Component rendered on the menu page for each item on the menu. Includes a picture of the item, the name, a description, and further details upon a mouse hover. Displays a "No Image" picture for recently added items or items that do not yet have pictures. Displays an "Out of Stock" message for items that are currently unavailable.
MenuPage	Displays the full menu separated into food, drinks, and add-ons. Each item, when clicked, reveals the *ItemPopUp* component with details about the item and a button to add the item to the user's cart. A side menu allows for quick navigation.
OrderPage	Displays the customer's current shopping cart and allows them to delete items from their cart. Allows users to place their order. Users are prompted with a confirmation popup containing an agreement, which they must accept to place their order. Afterwards, a popup explaining the next steps to retrieve their order is displayed.
FAQ	Allows users to view frequently asked questions and answers.
ContactUs	Form users can fill out to get in touch with the Expresso team.

BARISTA COMPONENTS

Name	Description
BaristaHeader	Navigation bar at the top of every page on the barista side. Allows navigation to the landing page, orders page, history page, inventory page, and barista logout. A confirmation popup appears on click of the logout button.
BaristaHistory	Retrieves all orders, complete or incomplete, from the current day, along with each order's placement time, item and add-ons, netid of the user who placed the order, and payment method. If there are currently no orders on the queue, a "No Orders!" message is displayed instead.
BaristaInventory	Shows all items that are available for selection on the client menu. Allows baristas to toggle the availability of each item, which then renders an "Out of Stock" message for that item on the menu. Also allows admins to add new items and delete items. If an admin attempts to delete an item, they will be prompted with a confirmation popup. A side menu allows for quick navigation.
BaristaLogin	Allows baristas and admins to login to the barista interface. All other barista pages reroute to this page if the user is not already logged in.
BaristaOrders	Displays all pending orders that have yet to be completed. Displays each order's time of placement, items, add-ons, and netid of the user who placed the item. Also allows baristas to select the "IN PROGRESS", "PENDING", and "COMPLETE" buttons. Orders fly off the queue once they are complete. Oldest orders sit at the top of the queue. The "COMPLETE" button becomes selectable once the order is paid for. Orders that are paid for with student charge are automatically marked with "PAID", whereas orders paid for in-store will remain "PENDING" until a barista marks them as "PAID".
AddItem	Renders the popup on the barista inventory page that allows admins to add new items. Admins enter different information for different categories of items.

SPECIAL CASES

If a client adds an item to their cart and the item is toggled out of stock before the client places their order, they will be presented with a popup telling them to remove that item from their cart to proceed with their order. After, the item will have an "Out of Stock" message on the main menu until toggled back on through the barista inventory page.

If a client who is not eligible for student charge (only undergraduates are eligible at the time of writing) attempts to place an order using student charge, they will be presented with a popup stating they are not eligible and must select in-store payment to proceed.

Baristas without admin privileges will be presented with an "Access denied" popup if they attempt to add or delete an item from the inventory.

If a client attempts to place an order with an empty shopping cart, they will be prompted with a popup asking them to add an item to their cart first.

If an admin attempts to add a new item without filling in all proper information, an informative error message will be displayed.

If the user does not fill in all fields on the Contact Us page or presents an email with an invalid email format, an informative error message will be displayed.

Design Challenges

SHOPPING CART

When designing our shopping cart, we first started having it set as a state in React that was sent around when the user navigated from page to page. This initial design, while functional for simpler iterations, meant that the cart's items were deleted if the user exited the tab. Instead of passing the shopping cart around as a state, we decided to change our approach and stored it in local storage with React. Through this change, users' shopping carts were saved in their browser, and they could re-navigate to them as long as the session had not ended (i.e. the user did not log out).

BARISTA LOGIN

For baristas, we had to design a different login system than CAS, which we used for customers. Since the baristas that are employed at Coffee Club varies, we didn't want to burden the managers by having to constantly change the permissions list. In addition, the barista interface is only meant to be logged into from one device (e.g. an iPad at the counter of the brick-and-mortar store), so we did not see the need to have multiple access accounts for the baristas. Thus, we had to make our own login system for the baristas.

We made a database table, valid_barista_users, that read in the usernames and passwords from a csv. The passwords were first salted and then encrypted using the SHA256 hash function. When a barista tries to login, we check whether the inputted username is stored in our database. If the username is valid, we take the inputted password, salt it, and then encrypt it using the same hash function. We then compare the hashed value of the inputted password and the valid password to see if they match.

STORING IMAGES

Our menu page displays images of Coffee Club's items. Initially we stored the images locally on the frontend. However, we wanted to give the Coffee Club administrators control of the application without having prior computer science knowledge. In this case. we wanted to design our system so that the Coffee Club admins could upload photos for new menu items in the future. Storing the images on the frontend was more technically challenging for the administrators, so we decided to store the images in the MySQL database as BLOBs. The server encodes the BLOB to a base-64 string, which is then passed over to the frontend. React renders the base-64 string as an image.

External Sources

BACKEND

For generating the daily report CSV, we adapted the code from Real Python. Also, for uploading images and converting it to a BLOB, we used code from PyNative. Finally, we adapted the LDAP code from Greg Blaha, the IT director for ODUS.

SERVER

For the email notifications, we used the email checking regex from Geek for Geeks to check whether an inputted email was valid or not. In addition we used the repository Mailgun from Github to design the HTML of our email receipt. Finally, we referenced the documentation pages for the Flask packages we installed.

FRONTEND

For CAS Client Authorization, we used CASClient.py provided by Professor Dondero in class. Besides using React-Semantic UI to better design our user interface, we used the layout to design the login form on the barista-side. Finally, we used the package google-map-react on our Location page.