

i4software Software Internship Project

Ryan Mohamed 🖐️

<https://restaurantrodeo.netlify.app>

▼ Restaurantrodeo 🤠

▼ Problems





- When using platforms like UberEats, DoorDash, Seamless, etc, one usually asks oneself or those nearby, "What do you one to eat?"
- In most cases the answer is "I don't know" or the atypical "Whatever you want to eat".
- These apps themselves usually show you the same few places you usually order from.
- When venturing off its hard to make a decision while scrolling through an abundance of new options with only rating as a guide.
- Alternatives like "Opentable" are tailored towards dine-in experiences and reservations in the area.
- Others like Yelp don't place much focus on one restaurant, leaving users to venture through numerous options and reviews.
- Other searches yield articles which are pre-populated with 1 or more individuals chosen information and opinion.

▼ Solutions





✅: Completed

Web App that queries restaurants to **order from** in the immediate area or others.

▼ User Requirements

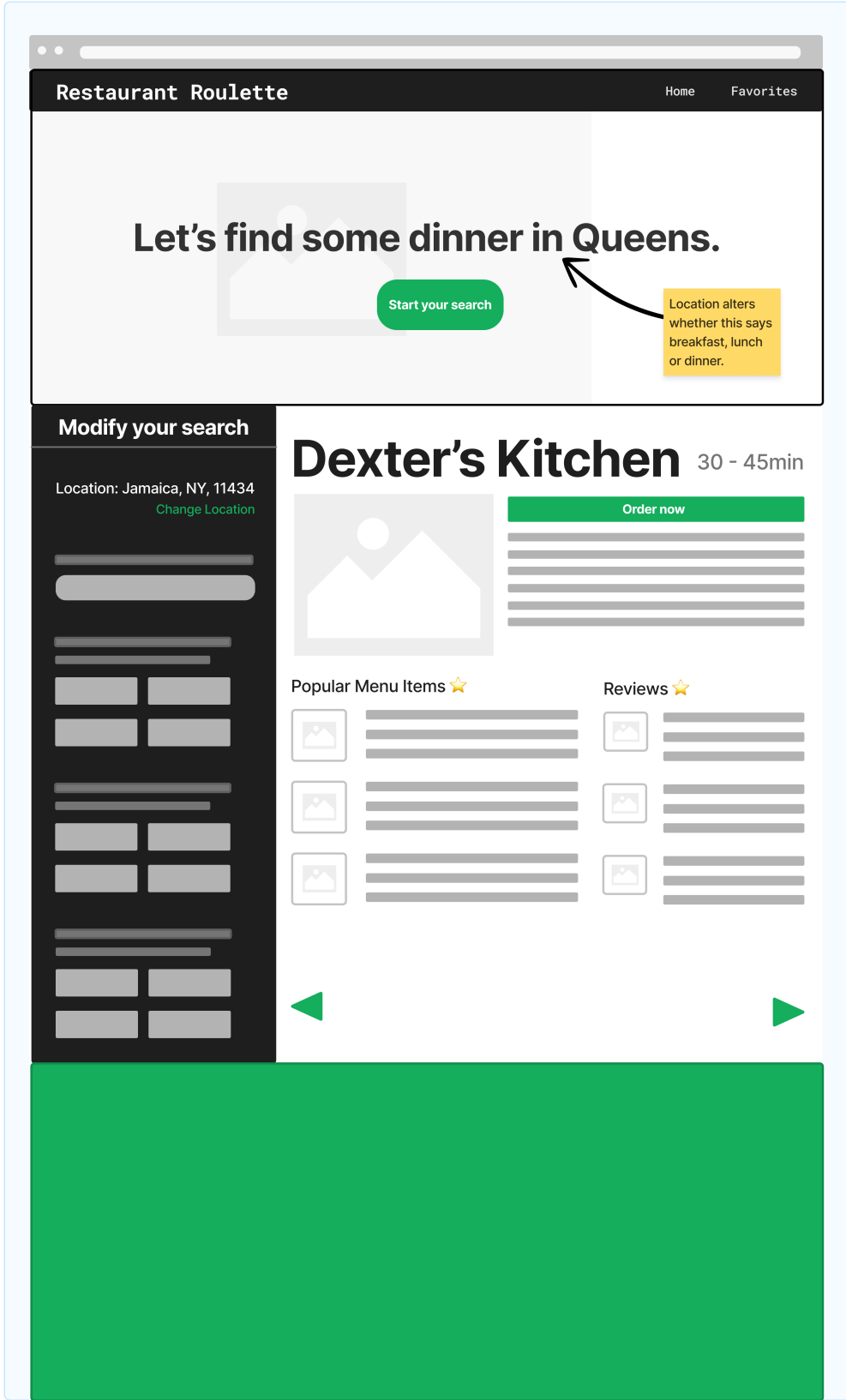
1. Should not require user to input location directly. 
2. Should instantly show popular items for a restaurant.
3. Should allow user to add food preferences. 
4. Should allow user to have a "favorites section". 
5. Should allow user to instantly find real reviews of a given restaurant. 
6. Should display estimated delivery times based on distance.

▼ Software Requirements

1. Should generate location based on IP. 
2. Should query local businesses API on server-side only. 
3. Should store user-specific information in browser to reduce API requests. 
4. Should cache queries to keep API request minimal. 

▼ **Design Rationale**

1. Should be naturally guide the user and inform them of what they can interact with on the site (intuitive). Consistent color theme.
2. Restaurant search should emphasize 1 restaurant at a time.
3. Modification panel should be easily accessible on any page.
4. Page should be responsive and adjust to mobile screens.
5. Favorites should be easy to find, and use.



▼ Development Process & Challenges Faced



When choosing a stack, I was looking for something performant, lightweight and still able to do both front-end and back-end communication to external APIs.

Next.js provides a framework for building full-stack applications with React and Node like syntax. The main point I wanted was SSR and SEO. Components need data only the server should be allowed to access, so we should fetch data and render components server-side and simply send to the client, i.e Next's perfect fit.

▼ Location API Request: (Client)

Before we start requesting from the Yelp API, whether that be directly on the client-side or proxied through the server, we need the user's **location**.

1. Before we start requesting from the Yelp API, whether that be directly on the client-side or proxied through the server, we need the user's **location**.
2. Many APIs provide this service. Our main concern is how do we manage this data? Some points:
3. It's client specific information so we can maintain some latitude and longitude state in React Context.
Since location can be found with some great free APIs we'll keep that logic neatly coupled client side.

1. `ipify` public API is used in our case for simple, and unlimited retrieval of client IPs. (As opposed to extracting client IP from requests to the web server. This method is more robust and utilizes the services provided by `ipify`)

2. `ip-api` public API is a great alternative to paid IP services. All we need is latitude and longitude (for future location based API calls), however `ip-api` provides some extra data for zero cost and with zero limitation. (This is an alternative to `navigator.geolocation` to emphasize API use)
1. Throughout the development process, I found that browsers like Safari block IP tracking websites on default. So as a fail-safe we explicitly ask for the user's location using `navigator.geolocation`
2. To decrease requests and the need to perform the above logic and improve usability, I store Location in the browsers cookies. If cookies are found that location is used across the app, otherwise we follow standard procedure.

▼ Yelp API Requests: (Server)

`yelp` public API provides the basis of our restaurant data with location directly influencing it. However its use is limited and authenticated with an API key. This communication should be kept away from the client entirely to ensure no requests to `yelp` in ill intent.

1. Client requests restaurants from Server.
2. Server requests `yelp` API.
3. Server parses and returns cleaned restaurant data as response to Client.
4. Client renders data as props on Page.

Yelp requires our application API Key, so we'll use the web app's server as a proxy when retrieving data so the client can never obtain our environment variables. This also aids in keeping our dynamic routes rendered on the server side.

Naturally the free tier of the `yelp` API employs a daily 500 request limit.

We are mainly performing 2 kinds of API requests to the `yelp` API.

1. **Initial search request** (SSR): yields a `SearchResponseType` JSON object with an array of `BusinessType` objects. The provides enough data to render business information on the server side and send it to the client. *1 request per search.*

1. **Supplemental Information Request:** (Proxied API route): Because the client renders one "page" of information from some `BusinessType[]` in a component at a time, we need only request the current business's supplemental information. *1 request per next page click.*

- a. Because we are requesting whenever the component's `page` state changes, we need to consider when we navigate backwards with `prev page` clicks.

- **Solution:** Check for cached restaurant data before making request on state change.

- b. Any user would naturally rapidly click through the `next page` button causing a ton of requests from our server to `yelp`.

- **Solution:** Delay/debounce the API request by `n` milliseconds and clear any previous requests that were waiting to be made. Requests are only made when the component has "sat" for `n` milliseconds.

▼ Google API Requests: (Server + Client)

1. Google `geocode` API provides latitude and longitude conversions for real addresses.

- a. Client initiates requests to web server. (Proxied API route)

- b. Web server makes requests to Google API.

- c. Google API returns data or null to web server.
 - d. Web server returns data for client to render and update UI.
2. Google `Map` JavaScript API used in `@react-google-maps/api`, not much work done by me except for event listeners and creating an API key. (Keep in mind, the client component requires an environment variable)
- a. IMPORTANT NOTE: Because the service is only accessible with a billing account, I had to add one. However the second it goes over 10 cents, I have set up a Pub-Sub topic which triggers a cloud function to disable billing entirely. Big thank you to the resources provided by (this video)
[\[https://www.youtube.com/watch?v=KiTg8RPpGG4&ab_channel=DataSlayer\]](https://www.youtube.com/watch?v=KiTg8RPpGG4&ab_channel=DataSlayer)

▼ Distance Matrix API [`distancematrix.ai`] (Server)

1. `distance-matrix` API provides distance matrix information (distance + duration) between coordinates or addresses for students for free (Hooray! 🤖) (We can replace the Google Geocode API with this one entirely, I discovered it a bit too late)
- a. Client initiates requests to web server. (Proxied API route)
 - b. Web server makes requests to Distance Matrix API.
 - c. Distance Matrix returns data or null to web server.
 - d. Web server returns data for client to render and update UI.
 - e. Incorporated in restaurant caching mechanism.
2. Google `Map` JavaScript API used in `@react-google-maps/api`, not much work done by me except for event listeners and

creating an API key. (Keep in mind, the client component requires an environment variable)

- a. IMPORTANT NOTE: Because the service is only accessible with a billing account, I had to add one. However the second it goes over 10 cents, I have set up a Pub-Sub topic which triggers a cloud function to disable billing entirely. Big thank you to the resources provided by (this video)
[\[https://www.youtube.com/watch?v=KiTg8RPpGG4&ab_channel=DataSlayer\]](https://www.youtube.com/watch?v=KiTg8RPpGG4&ab_channel=DataSlayer)

▼ Testing Strategies

Many of the tests for this project were done manually in the app or using applications like Insomnia, as opposed to using testing frameworks like Jest or Cypress. The focus on my tests were failed API requests, bad user input or forced navigations. The results of many of my tests was to make sure data strongly conforms to what is expected, preventative maintenance!

Since location is the foundation of the app and future API request heavy testing was done to ensure errors are caught, handled and rendered beautifully in the UI. Any non 200 status will generate an error and if the received data does not conform to type checking an error will be thrown.

I wanted to eliminate forced user navigations from the get go, and decided to use location cookies as a requirement to get to any sub route. The user needs to have a location saved to cookies before they can request any sub-route. This comes into play later on in challenges faced.

Yelp errors were easier to handle since online documentation provided strong type information and requests yielded specific status code and error messages. Since the data retrieved from Yelp is simply used as props for SSR components and in a proxied API route (which itself returns data to be used as component props), errors will yield a Wireframe and defined props will yield a full Component. This made manual testing extremely fast to decipher and error handling to be rather clean.

However a considerable amount of testing was done on the number of API requests being made to verify the intended number of requests are being made (1 per search, 1 per restaurant view for > 1500ms). This definitely could have been replaced with tests written in Cypress or Jest to check the number of times a function or mockAPI is called on mount and pagination.

The last bit of testing came in Google API requests. Since Google Cloud Platforms has many options such as limiting who can use an API key (referrer) or the number of APIs the API key can access, I didn't have much to worry about when it came to security (I hope). However if the API Key ever expires or is disabled due to a lacking a billing account (will be removed for > \$0.10), the error is handled by the Map component internally and the map will render the error.

However for my proxied API endpoint that requests another Google API, errors can absolutely happen. These can stem from input error, API failure, etc. So the results of these test yield additions of type checking and data conformity in the proxied API endpoint before any request is made to Google. Errors from Google are passed to the client and handled appropriately.

Basic component tests done with:

@react-testing-library

@testing-library/jest-dom

@testing-library/jest

Very fun working on this!

Tasks like these get me so excited at the possibility of working alongside leading companies like i4software! Looking forward to your feedback!