Restaurant Application Capstone Presentation

Erik Slagter

Project Links

GitHub Repository
https://github.com/erik06/MIT-capstone

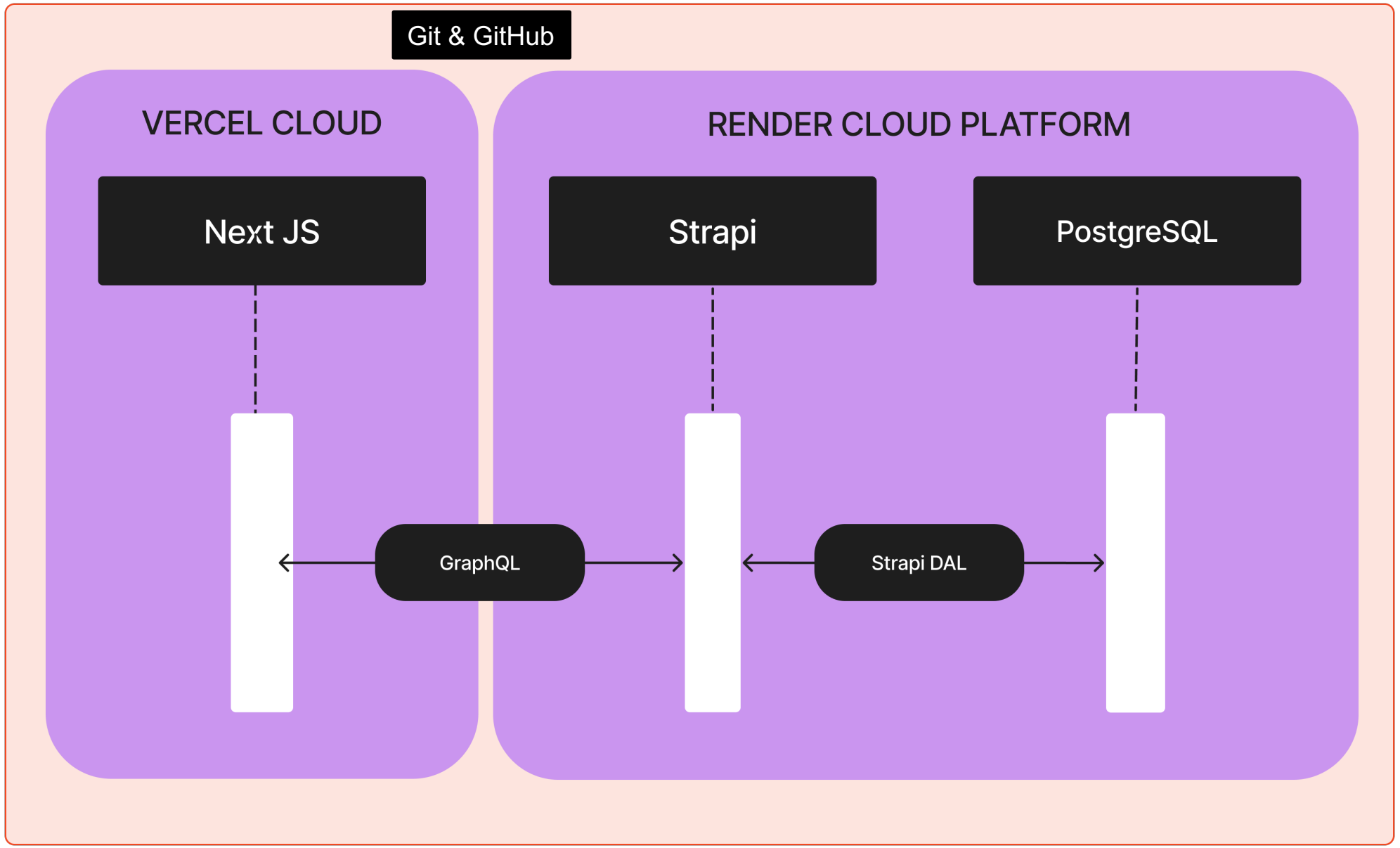
Vercel Front End (Next JS)
https://mit-capstone.vercel.app

Strapi Backend (Render Cloud)
https://strapi-75qo.onrender.com/admin

GraphQL Playground (Render Cloud)
https://strapi-75qo.onrender.com/graphql

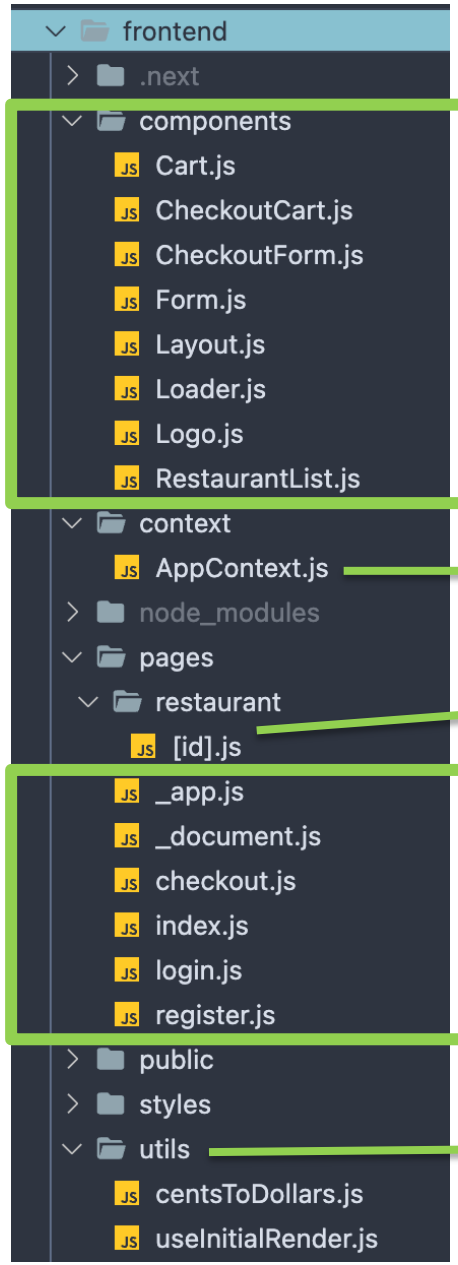
Part 1: Front-End Architecture, Authentication, And App Diagram

Application Overview Diagram



xPRO

Front-End Architecture – Next JS



Small pieces of code that are shared and amongst pages and other areas of the app. Some components are simple, like “Logo.js” which just contains the SVG logo for “Chicago Chow” verses a more involved component like ”CheckoutForm.js” which contains all the logic and validation for the cart and checkout system.

This app utilizes React Context to store all the information the user generates during their time on the application. This logic sets cookies for storing information such as cart contents and the users JSON Web Token.

Restaurant detail pages are dynamically rendered via Next JS with a Graph QL call to the Strapi database via restaurant ID.

These files make up the core of the app, the high-level layout and the homepage of the app. Here too is the login and register pages which allow user login-logout.

Finally, the utils folder contains a few helpful utilities used throughout the application. For example, “centsToDollars.js” is responsible for calculating cents to dollars and making the prices easier to read.

Authentication

```
You, 19 hours ago | 1 author (You)
import { useState } from "react";    Parsing error: Cannot find module 'next/babel'⏎Require
import { useRouter } from "next/router";  89.3k (gzipped: 26.4k)
import { useAppContext } from "@/context/AppContext";
import { gql, useMutation } from "@apollo/client";  11.5k (gzipped: 4.5k)
import Cookie from "js-cookie";  1.7k (gzipped: 848)

import Form from "@/components/Form";
import Loader from "@/components/Loader";

const REGISTER_MUTATION = gql`
  mutation Register($username: String!, $email: String!, $password: String!) {
    register(
      input: { username: $username, email: $email, password: $password }
    ) {
      jwt
      user {
        username
        email
      }
    }
  }
`;

export default function RegisterRoute() {
  const { setUser } = useAppContext();
  const router = useRouter();

  const [formData, setFormData] = useState({ email: "", password: "" });
  const [registerMutation, { loading, error }] = useMutation(REGISTER_MUTATION);

  const handleRegister = async () => {
    const { email, password } = formData;
    const { data } = await registerMutation({
      variables: { username: email, email: email, password },
    });
    if (data?.register.user) {
      setUser(data.register.user);
      router.push("/");
      Cookie.set("token", data.register.jwt);
      alert("You have successfully logged in and created an account!");
    }
  };

  if (loading) return <Loader />;

  return (
    <Form
      title="Sign Up"
      buttonText="Sign Up"
      formData={formData}
      setFormData={setFormData}
      callback={handleRegister}
      error={error}      You, 2 days ago • dishes and login
    />
  );
}
```

Registration

The code shown is for user registration with a Strapi backend within a Next.js application. Utilizing the Apollo Client, it sends GraphQL mutation requests for user registration. When a user fills out the form with their email and password and clicks "Sign Up", the `**handleRegister**` function is invoked. This function sends the user's email and password as variables in the `**REGISTER_MUTATION**` GraphQL mutation to the Strapi backend. If the registration is successful, Strapi returns a JSON Web Token (JWT) and user details. This JWT is then stored as a "token" in a browser cookie to maintain the user's authenticated state, while the user details are saved in the application's context using the `**setUser**` method from `**useAppContext**`. After successful registration, the user is redirected to the homepage and presented with a success alert. If the mutation request is still being processed, a loader is shown, and if there's an error, it can be displayed using the `**error**` prop in the `**Form**` component.

Authentication

```
You, 19 hours ago | 1 author (You)
import { useState } from "react";    Parsing error: Cannot find module 'next/babel
import { useRouter } from "next/router";  89.3k (gzipped: 26.4k)
import { useAppContext } from "@/context/AppContext";
import { gql, useMutation } from "@apollo/client";  11.5k (gzipped: 4.5k)
import Cookie from "js-cookie";  1.7k (gzipped: 848)

import Form from "@/components/Form";
import Loader from "@/components/Loader";

const LOGIN_MUTATION = gql`
  mutation Login($identifier: String!, $password: String!) {
    login(input: { identifier: $identifier, password: $password }) {
      jwt
      user {
        username
        email
      }
    }
  }
`;

export default function LoginRoute() {
  const { setUser } = useAppContext();
  const router = useRouter();
      You, 2 days ago • dishes and login
  const [formData, setFormData] = useState({ email: "", password: "" });
  const [loginMutation, { loading, error }] = useMutation(LOGIN_MUTATION);

  const handleLogin = async () => {
    const { email, password } = formData;
    const { data } = await loginMutation({
      variables: { identifier: email, password },
    });
    if (data?.login.user) {
      setUser(data.login.user);
      Cookie.set("token", data.login.jwt);
      router.push("/");
      alert("You have successfully logged in!");
    }
  };

  if (loading) return <Loader />;

  return (
    <Form
      title="Login"
      buttonText="Login"
      formData={formData}
      setFormData={setFormData}
      callback={handleLogin}
      error={error}
    />
  );
}
```

Login

This code showcases the user authentication (login) with a Strapi backend in a Next.js application. It leverages the Apollo Client to send a GraphQL mutation request for logging in. Users input their email and password in the form and upon clicking "Login", the **handleLogin** function is called. This function forwards the user's email as the identifier and the password as variables to the **LOGIN_MUTATION** GraphQL mutation directed at the Strapi backend. Once authenticated successfully, Strapi replies with a JSON Web Token (JWT) and user data. The JWT is stored as a "token" cookie to maintain the user session, and the user information is updated in the app's context with **setUser** from **useAppContext**. After a successful login, users are rerouted to the application's main page and are greeted with a success notification. While the mutation is in progress, a loader is displayed, and any errors encountered during the process can be reflected through the **error** prop in the **Form** component.

Stripe

CheckoutForm.js 1

frontend > components > CheckoutForm.js > CheckoutForm >

```
You, 2 days ago | 1 author (You)
import React, { useState } from "react";    Parsing
import Cookie from "js-cookie";
import { client } from "@/pages/_app.js";
import { gql } from "@apollo/client";
import { CardElement, useStripe, useElements } from
import { useAppContext } from "@/context/AppContext
import { useRouter } from "next/router";
import { useInitialRender } from "@/utils/useInitia

const options = {
  style: {
    base: {
      fontSize: "32px",
      color: "#52a635",
      "::placeholder": {
        color: "#aab7c4",
      },
    },
    invalid: {
      color: "#9e2521",
    },
  },
};

const INITIAL_STATE = {
  address: "",
  city: "",
  state: "",
  error: null,
};

export default function CheckoutForm() {
  const [data, setData] = useState(INITIAL_STATE);
  const [loading, setLoading] = useState(false);
  const { user, cart, resetCart, setShowCart } = us

  const initialRender = useInitialRender();

  const stripe = useStripe();
  const elements = useElements();
  const router = useRouter();

  if (!initialRender) return null;

  function onChange(e) {
```

checkout.js 1

frontend > pages > checkout.js > ...

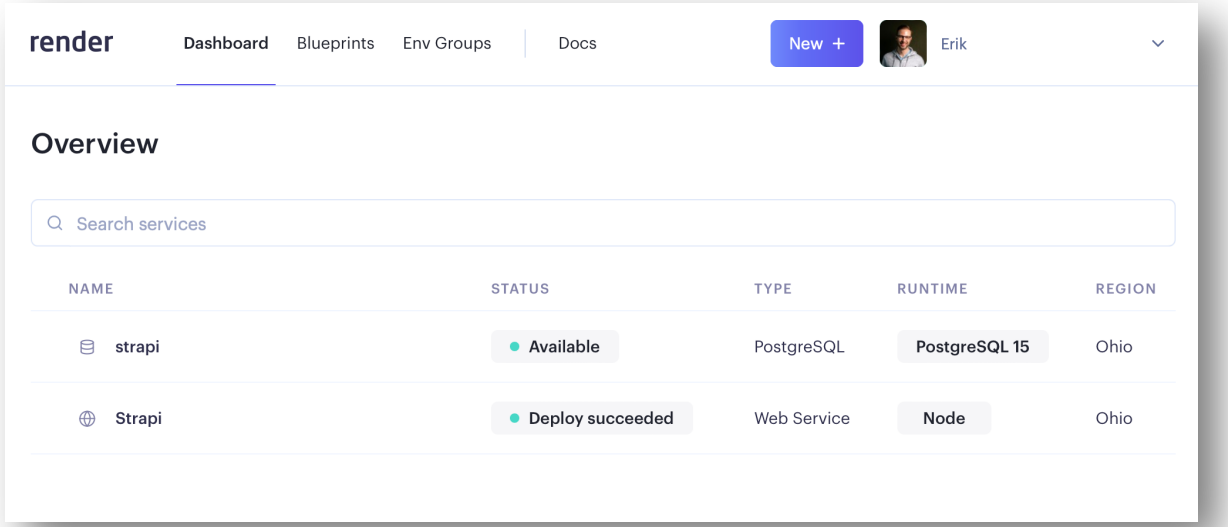
```
You, 2 days ago | 1 author (You)
import { Elements } from "@stripe/react-stripe-js"
import { loadStripe } from "@stripe/stripe-js";
import { useInitialRender } from "@/utils/useInitia
import CheckoutForm from "@/components/CheckoutFor
import CheckoutCart from "@/components/CheckoutCart
const stripePromise = loadStripe("pk_test_TYooMQau

export default function Checkout() {
  const initialRender = useInitialRender();
  if (!initialRender) return null;

  return (
    <section className="ooo">
      <div className="ooo">
        <div className="ooo">
          <CheckoutCart />
        </div>
        <div className="ooo">
          <Elements stripe={stripePromise}>
            <CheckoutForm />
          </Elements>
        </div>
      </div>
    </section>
  );
}
```

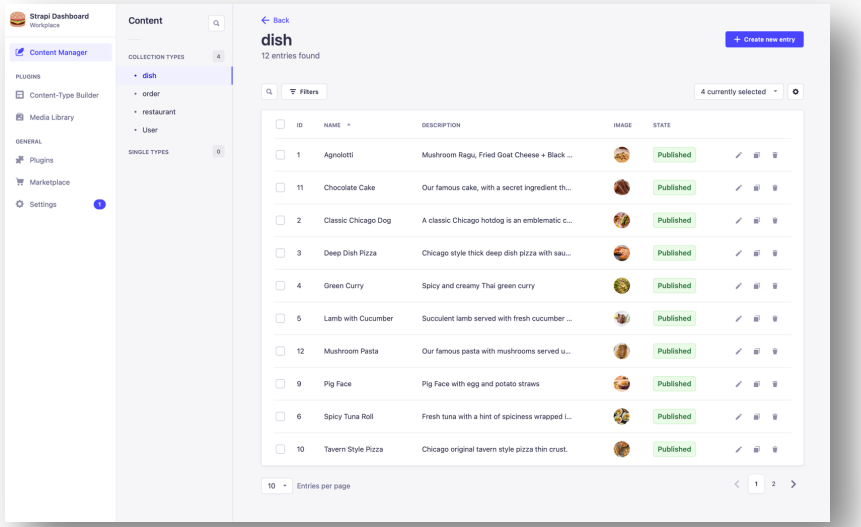
In this implementation, Stripe is integrated into the shopping cart functionality to handle payment processing. The `**Checkout.js**` file initializes Stripe using the `**loadStripe**` function from `**@stripe/stripe-js**` and provides the Stripe context via the `**Elements**` component from `**@stripe/react-stripe-js**` to any child components, particularly the `**CheckoutForm**` component where payment details are collected.

Within the `**CheckoutForm.js**` file, Stripe's `**useStripe**` and `**useElements**` hooks are employed to extract necessary utilities for creating payment tokens and interfacing with the card elements. Users can fill in their shipping information (address, city, state) and provide card details through Stripe's `**CardElement**`, which encapsulates a UI for credit/debit card input while ensuring PCI compliance. On form submission, the `**submitOrder**` function retrieves the entered card information, uses Stripe's utilities to create a token representing that card, and sends this token along with other order details (like address and cart items) to the server using a GraphQL mutation. On the server side (not shown), Stripe would use this token to charge the user. The application also has error handling in place to notify users about any issues during this process, such as missing address fields or card errors. Once the transaction is successful, the user is alerted, the cart is reset, and they are redirected to the homepage.

Part 2: Database And API

Database





What data in the application is persistent and stored in your database?

In my application, the persistent data stored in the database encompasses several key entities: **users**, **orders**, **restaurants**, and **dishes**. Additionally, the database houses **images** for both restaurants and their respective dishes.

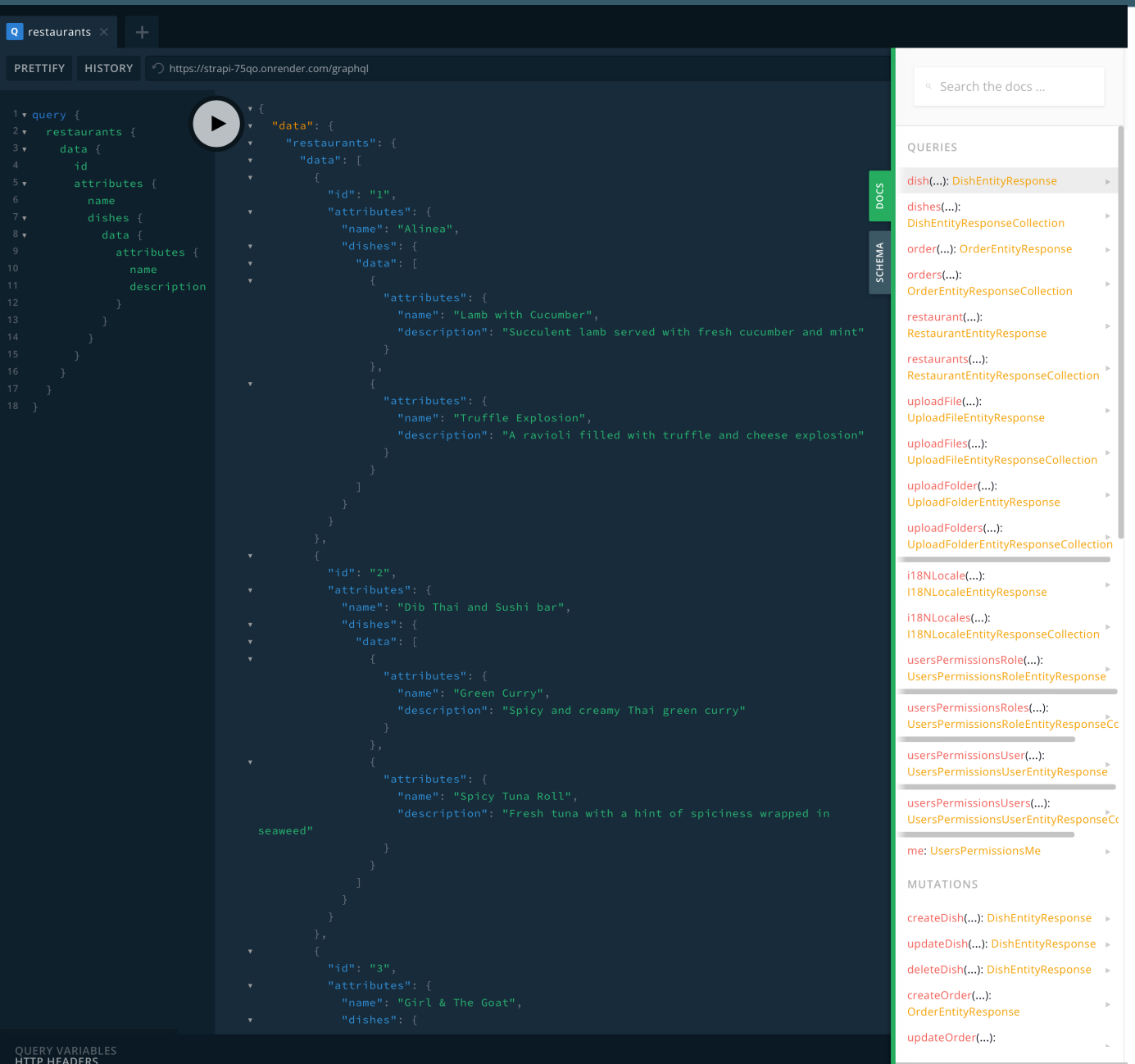
Discuss any new database features you've developed as well as the design decisions you made when implementing those features. What did you refactor from the starter code?

While I haven't introduced any new database-specific features, there's potential for future enhancements, especially the integration of restaurant reviews from authenticated users. This addition could elevate the user experience by providing a more comprehensive understanding of each dining option. I heavily refactored the shopping cart, moving it to a new place that doesn't overlap any content. I also made the toggle cart button very clear.

What challenges did you encounter? How did you overcome those challenges?

On the technical side, I faced challenges during deployment, particularly with integrating both Strapi and the associated database within a cloud environment. Ensuring the flawless display and access to the stored images was another complexity. The migration of data from my local Strapi setup to the cloud was perhaps one of the most intricate tasks. My solution involved leveraging terminal commands to export Strapi data from the local setup. Following this, I employed the "scp" command, enabling me to seamlessly transfer the files to the remote cloud instance.

API



GraphQL

I used GraphQL for my project, here is a sample query from the GraphQL playground, along with the Docs section shown, which displays all of the possible queries and mutations available in this API.

I highlight the main endpoints on the next slide.

MIT xPRO

API - Endpoints

1. **dish**
   - **Role**: This endpoint would be used to fetch or modify details about a single dish item.
   - **Possible Operations**:
     - `query`: Retrieve the details of a specific dish based on a dish ID or other criteria.
     - `mutation`: Create a new dish, update an existing dish's details, or delete a dish.

2. **dishes**
   - **Role**: Fetch a list of dishes, possibly with filtering, sorting, or paging capabilities.
   - **Possible Operations**:
     - `query`: Retrieve a list of all dishes or those that match certain criteria.

3. **order**
   - **Role**: Handle operations for a single order. This could involve fetching order details, updating order status, or making modifications to the items in the order.
   - **Possible Operations**:
     - `query`: Retrieve the details of a specific order based on an order ID.
     - `mutation`: Place a new order, update an existing order's details or status, or delete an order.

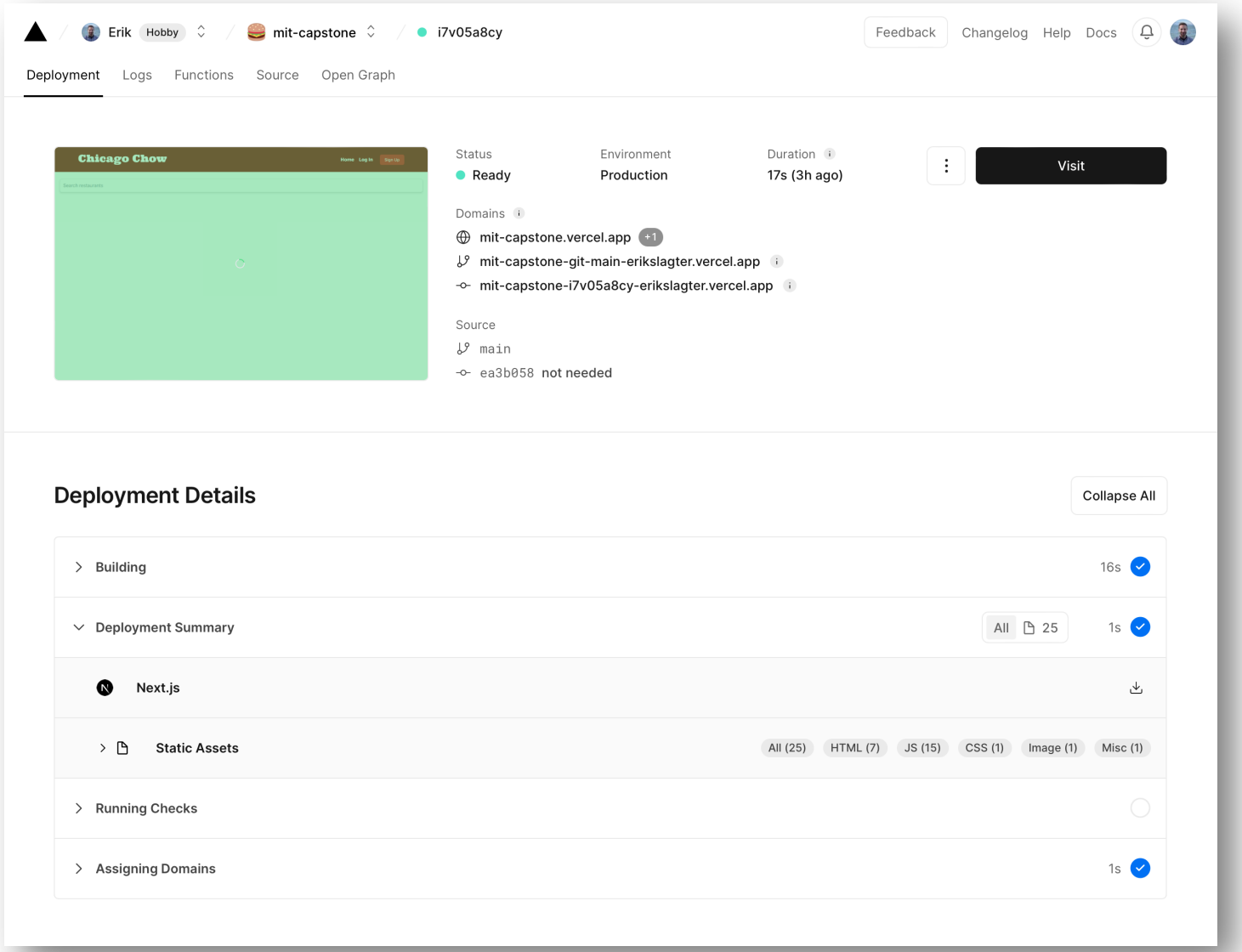
4. **orders**
   - **Role**: Retrieve a collection of orders. Like `dishes`, this might also support filtering, sorting, and paging.
   - **Possible Operations**:
     - `query`: Retrieve a list of all orders or those matching certain criteria.

5. **restaurant**
   - **Role**: Manage the information and operations for a single restaurant.
   - **Possible Operations**:
     - `query`: Fetch the details of a specific restaurant based on its ID or other criteria.
     - `mutation`: Add a new restaurant, update details of an existing restaurant, or delete a restaurant.

6. **restaurants**
   - **Role**: Retrieve information about multiple restaurants. Again, this could support capabilities like filtering, sorting, or paging.
   - **Possible Operations**:
     - `query`: Get a list of all restaurants or those that match specific criteria.

Part 3: Deployment, Additional Features, App Demonstration, Reflection

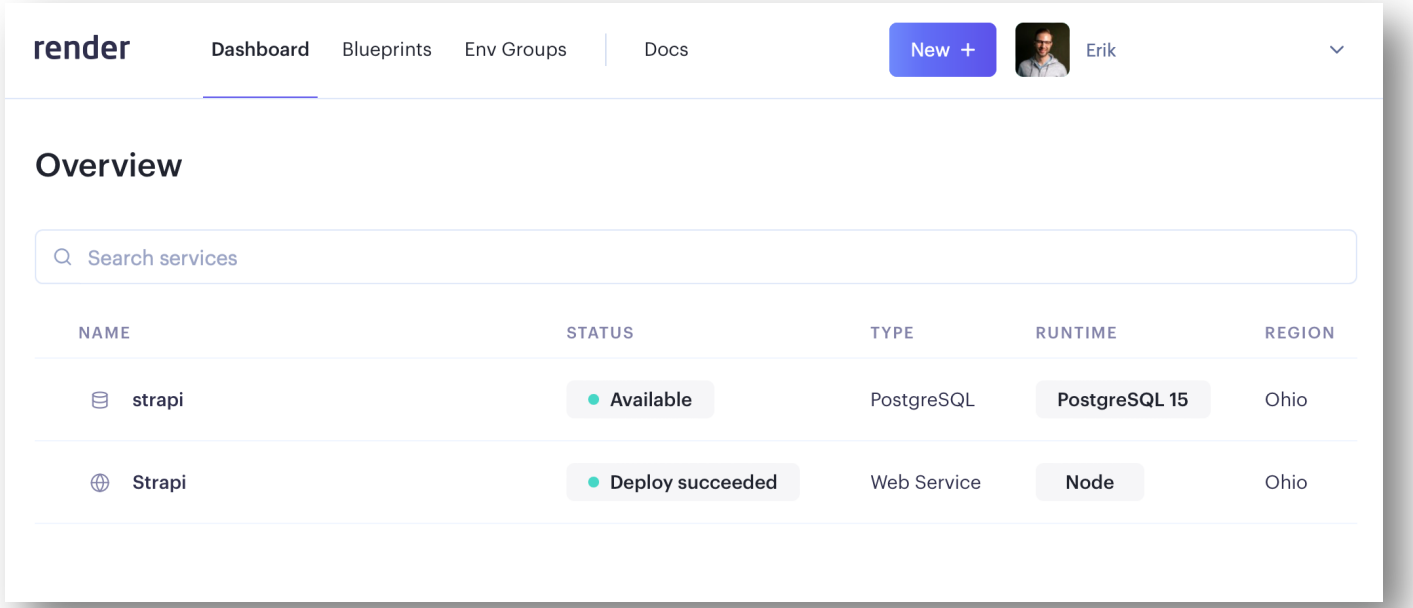
Deployment

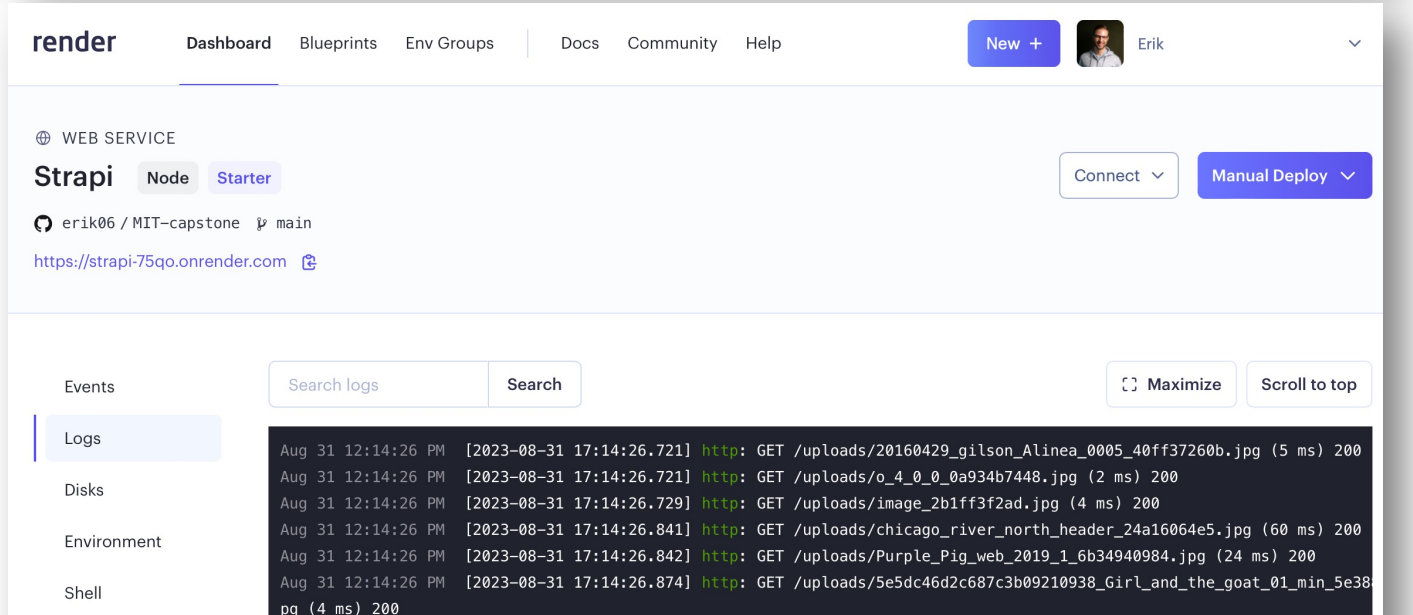


Front End – Next JS on Vercel

I deployed my Next.js app on Vercel through a streamlined and efficient process. First, I ensured that my Next.js application was ready in a Git repository, since Vercel integrates seamlessly with version control platforms like GitHub, GitLab, and Bitbucket. Once my app was in a repository, I signed up and logged in to Vercel, then connected my Git account. After connecting, Vercel displayed a list of my repositories. I selected the repository containing my Next.js app. Vercel automatically detected that it was a Next.js project and provided me with default build settings. I confirmed these settings, and Vercel deployed my app, giving me a live URL. Every time I pushed to the connected branch afterward, it triggered automatic deployments, ensuring my live site was always up-to-date with my latest code changes.

Deployment





Back End – Strapi & PostgreSQL on Render

I deployed my Strapi API on the Render cloud platform; I began by preparing my Strapi project for deployment. This involved ensuring that the database configurations were set up correctly, using environment-specific variables, and then pushing my Strapi codebase to my Git repository. With my Strapi project in a Git repository, I proceeded to Render's dashboard and created a new web service. After linking the repository, I set up the necessary environment variables for my database and other configurations (and pointed Render to my “backend” folder). Render automatically detected the build and start commands from my `package.json`. Once everything was set up, I triggered the build and deployment process. Within minutes, my Strapi API was live on Render, accessible via a Render-generated URL. I also made sure to set up a PostgreSQL database on Render for persistence, ensuring that my data remained secure and available.

Deployment

How did you deploy your app?
I deployed my app on both Vercel and Render Cloud, leveraging the benefits of cloud hosting. The site is rebuilt on each git commit.

Where is it hosted?
It's hosted on both Vercel and Render Cloud. Utilizing Render for Strapi (Node JS) and a managed PostgreSQL service.

Did you use Docker?
While Docker is a common choice for many, I opted not to use it for this particular deployment because my application is hosted on many different providers.

What changes did you make to the project in order to deploy it?
One of the primary changes I made was ensuring that the URLs were correctly configured to allow seamless communication between the front-end Next.js application and the backend services.

What challenges did you experience?
One of the main challenges I encountered was ensuring the URLs allowed the front-end Next.js application to communicate effectively with the Strapi and PostgreSQL backend services.

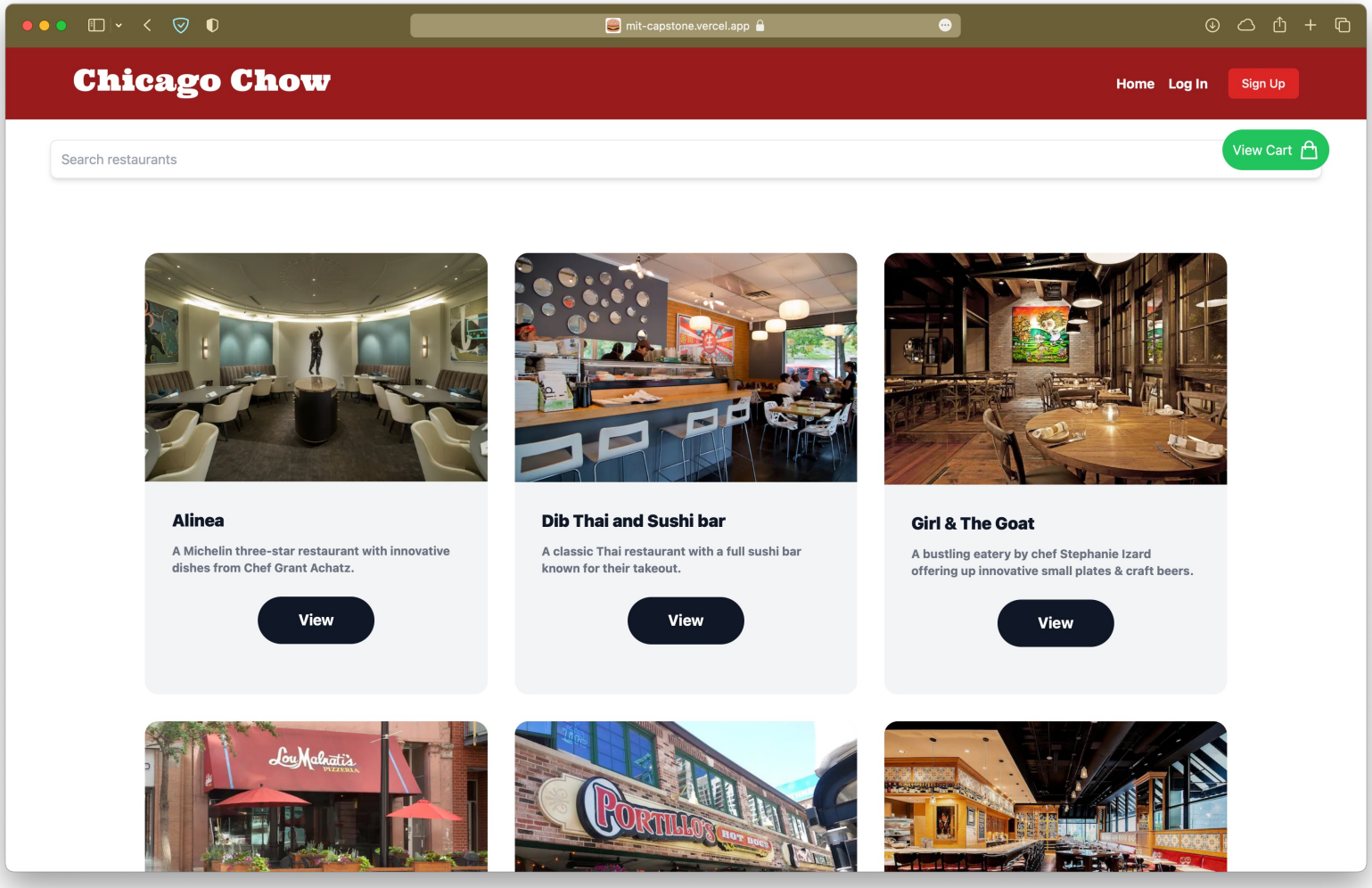
How did you overcome those challenges?
To resolve this issue, I thoroughly reviewed the documentation and made the necessary adjustments to ensure uniformity in the URLs.

Additional Features

In the time allocated, I wasn't able to implement any extra features. However, I identified that a valuable enhancement would be a dedicated section for the registered users to share their reviews on restaurants and specific dishes within our application.

I anticipated potential issues in this endeavor, especially when it comes to establishing the right API routes for this influx of data. Moreover, ensuring that my API calls accurately targeted the corresponding dish and restaurant via GraphQL would present its own set of complexities. This foresight would help me prepare for potential challenges down the road.

Application Demonstration – Create Account, Log In, Cart Functions



https://mit-capstone.vercel.app

Reflection

If I were to embark on this project today, while the core coding and structure might remain largely unchanged, my approach to deployment would be significantly different. One of the primary lessons I've learned from this experience was the importance of deploying earlier in the development process. I waited until the application was nearly complete before initiating deployment, only to discover that this phase presented the most significant challenges. Deploying earlier would have allowed me to identify and rectify any deployment-related issues incrementally, making the process smoother and more efficient.

Regarding additional functionalities, I've previously touched upon the inclusion of reviews and ratings for dishes and restaurants, emphasizing their potential value to users. Beyond that, I've been pondering the incorporation of a personalized profile page. This feature would provide a consolidated view for logged-in users, showcasing all their past orders. Such a feature would not only enhance the user experience by offering a quick glance at their order history but also foster a more tailored and engaging platform for the clientele.


xPRO