



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>

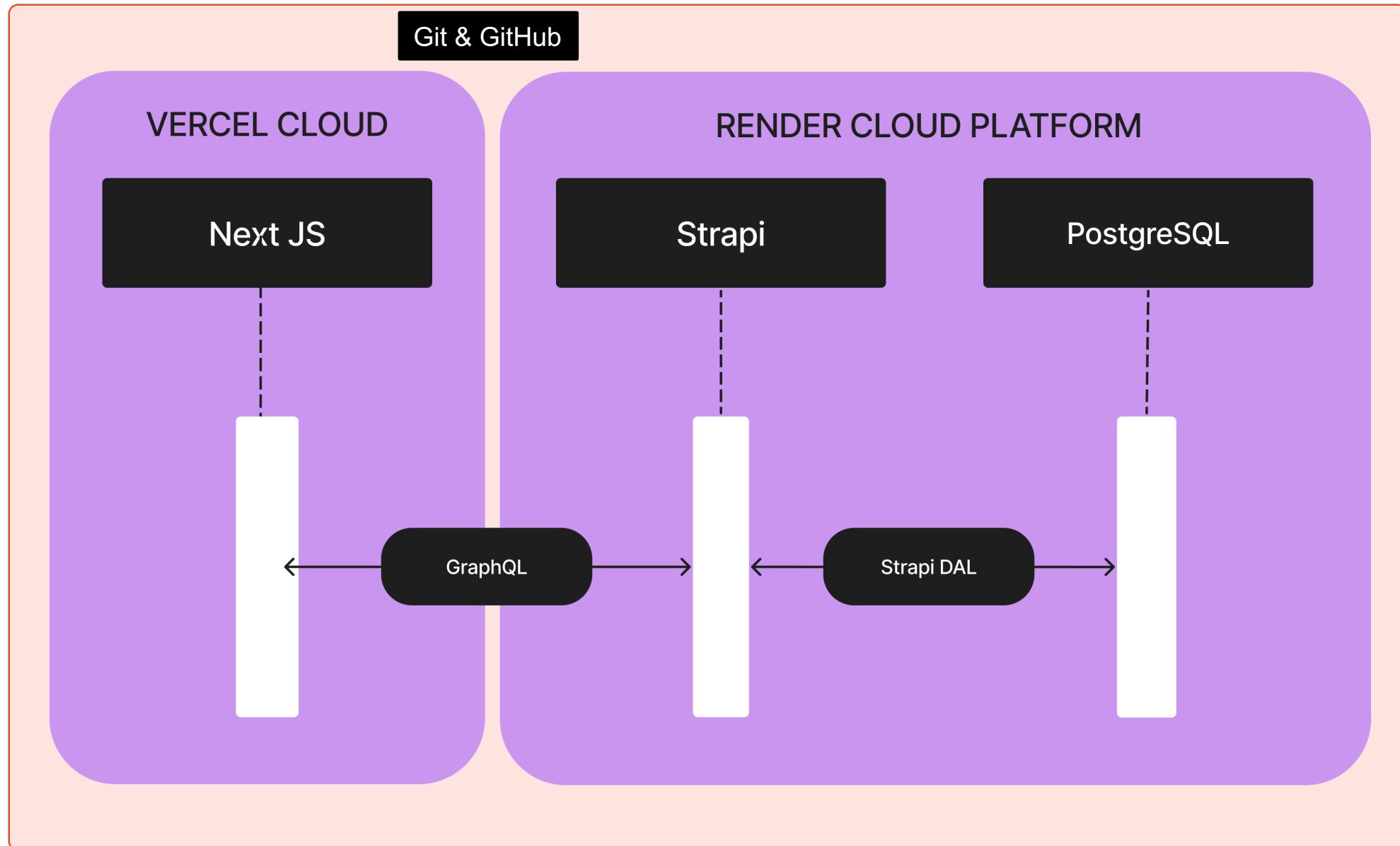
GraphQL Playground (Render Cloud)

<https://strapi-75qo.onrender.com/graphq>

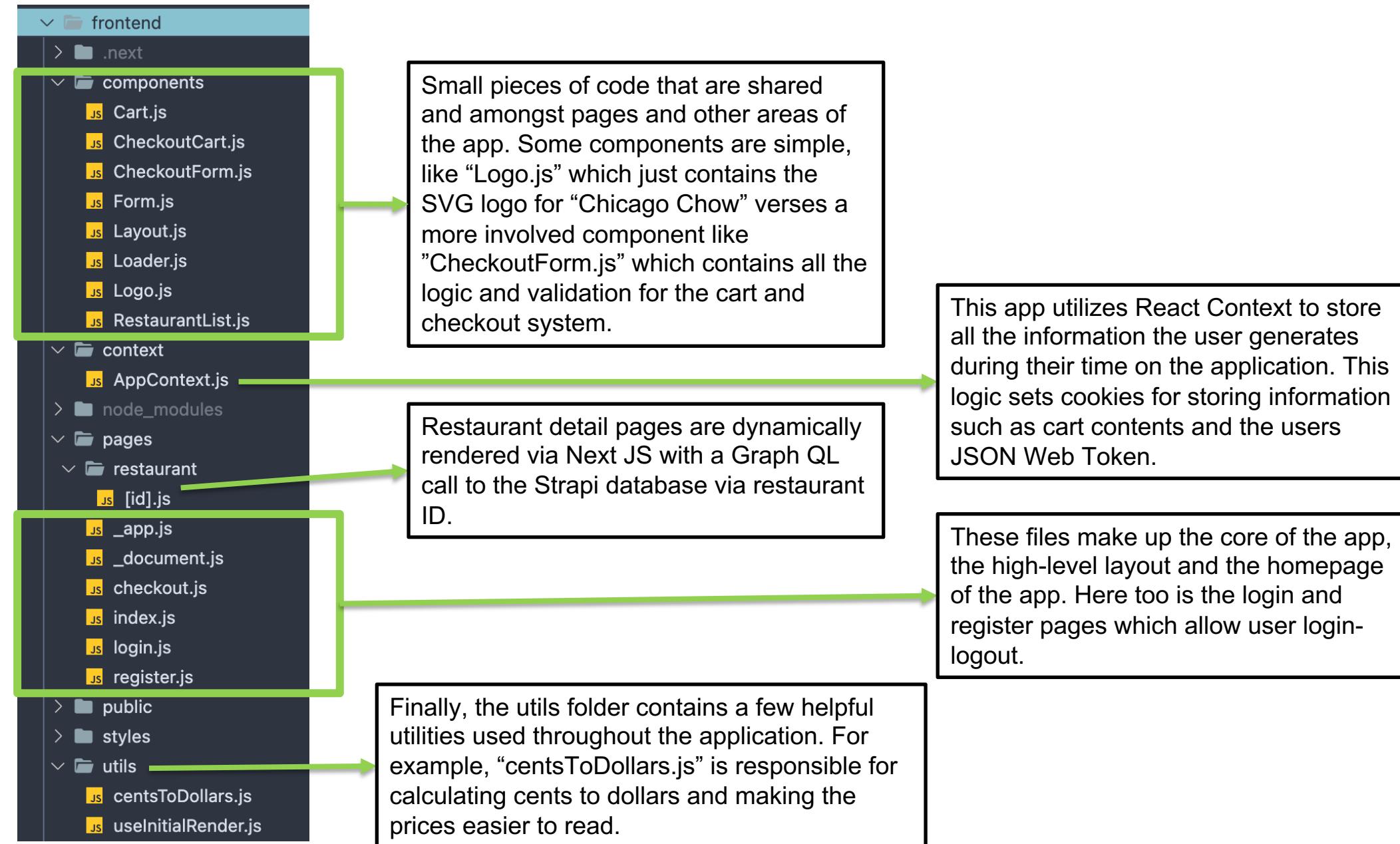


Part 1: Front-End Architecture, Authentication, And App Diagram

Application Overview Diagram



Front-End Architecture – Next JS



Authentication

```
You, 19 hours ago | 1 author (You)
1 import { useState } from "react";    Parsing error: Cannot find module 'next/babel'@Require
2 import { useRouter } from "next/router";  89.3k (gzipped: 26.4k)
3 import { useAppContext } from "@context/AppContext";
4 import { gql, useMutation } from "@apollo/client";  11.5k (gzipped: 4.5k)
5 import Cookie from "js-cookie";  1.7k (gzipped: 848)
6
7 import Form from "@/components/Form";
8 import Loader from "@/components/Loader";
9
10 const REGISTER_MUTATION = gql` 
11   mutation Register($username: String!, $email: String!, $password: String!) {
12     register(
13       input: { username: $username, email: $email, password: $password }
14     ) {
15       jwt
16       user {
17         username
18         email
19       }
20     }
21   }
22 `;
23
24 export default function RegisterRoute() {
25   const { setUser } = useAppContext();
26   const router = useRouter();
27
28   const [formData, setFormData] = useState({ email: "", password: "" });
29   const [registerMutation, { loading, error }] = useMutation(REGISTER_MUTATION);
30
31   const handleRegister = async () => {
32     const { email, password } = formData;
33     const { data } = await registerMutation({
34       variables: { username: email, email: email, password },
35     });
36     if (data?.register.user) {
37       setUser(data.register.user);
38       router.push("/");
39       Cookie.set("token", data.register.jwt);
40       alert("You have successfully logged in and created an account!");
41     }
42   };
43
44   if (loading) return <Loader />;
45
46   return (
47     <Form
48       title="Sign Up"
49       buttonText="Sign Up"
50       formData={formData}
51       setFormData={setFormData}
52       callback={handleRegister}
53       error={error}  You, 2 days ago * dishes and login
54     />
55   );
56 }
```

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)
1 import { useState } from "react";    Parsing error: Cannot find module 'next/babel
2 import { useRouter } from "next/router";  89.3k (gzipped: 26.4k)
3 import { useAppContext } from "@context/AppContext";
4 import { gql, useMutation } from "@apollo/client";  11.5k (gzipped: 4.5k)
5 import Cookie from "js-cookie";  1.7k (gzipped: 848)
6
7 import Form from "@/components/Form";
8 import Loader from "@/components/Loader";
9
10 const LOGIN_MUTATION = gql`  
11   mutation Login($identifier: String!, $password: String!) {  
12     login(input: { identifier: $identifier, password: $password }) {  
13       jwt  
14       user {  
15         username  
16         email  
17       }  
18     }  
19   }  
20 `;  
21
22 export default function LoginRoute() {  
23   const { setUserData } = useAppContext();  
24   const router = useRouter();
25   You, 2 days ago * dishes and login
26   const [formData, setFormData] = useState({ email: "", password: "" });
27   const [loginMutation, { loading, error }] = useMutation(LOGIN_MUTATION);
28
29   const handleLogin = async () => {
30     const { email, password } = formData;
31     const { data } = await loginMutation({
32       variables: { identifier: email, password },
33     });
34     if (data?.login.user) {
35       setUserData(data.login.user);
36       Cookie.set("token", data.login.jwt);
37       router.push("/");
38       alert("You have successfully logged in!");
39     }
40   };
41
42   if (loading) return <Loader />;
43
44   return (
45     <Form
46       title="Login"
47       buttonText="Login"
48       formData={formData}
49       setFormData={setFormData}
50       callback={handleLogin}
51       error={error}
52     />
53   );
54 }
```

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

The screenshot shows a code editor with two tabs open: 'CheckoutForm.js' and 'checkout.js'. Both files are located in a 'frontend' directory.

CheckoutForm.js:

```
You, 2 days ago | 1 author (You)
1 import React, { useState } from "react";      Parsing
2 import Cookie from "js-cookie";
3 import { client } from "@/pages/_app.js";
4 import { gql } from "@apollo/client";
5 import { CardElement, useStripe, useElements } from
6 import { useAppContext } from "@/context/AppContext"
7 import { useRouter } from "next/router";
8 import { useInitialRender } from "@/utils/useInitia
9
10 const options = {
11   style: {
12     base: {
13       fontSize: "32px",
14       color: "#52a635",
15       "::placeholder": {
16         color: "#aab7c4",
17       },
18     },
19     invalid: {
20       color: "#9e2521",
21     },
22   },
23 };
24
25 const INITIAL_STATE = {
26   address: "",
27   city: "",
28   state: "",
29   error: null,
30 };
31
32 export default function CheckoutForm() {
33   const [data, setData] = useState(INITIAL_STATE);
34   const [loading, setLoading] = useState(false);
35   const { user, cart, resetCart, setShowCart } = us
36
37   const initialRender = useInitialRender();
38
39   const stripe = useStripe();
40   const elements = useElements();
41   const router = useRouter();
42
43   if (!initialRender) return null;
44
45   function onChange(e) {
46     const { name, value } = e.target;
47     const updatedData = { ...data, [name]: value };
48     setData(updatedData);
49
50     if (name === "cardNumber") {
51       const cardNumber = updatedData.cardNumber.replace(
52         /\D/g, // Remove non-digit characters
53         ""
54       );
55       const card = elements.createCardNumberInput();
56       card.value = cardNumber;
57       stripe.createToken(card).then((token) => {
58         updatedData.token = token.id;
59         setData(updatedData);
60       });
61     }
62
63     if (name === "cvv") {
64       const cvv = updatedData.cvv.replace(/\D/g, ""); // Remove non-digit characters
65       const card = elements.createCardCvcInput();
66       card.value = cvv;
67       stripe.createToken(card).then((token) => {
68         updatedData.token = token.id;
69         setData(updatedData);
70       });
71     }
72
73     if (name === "expMonth" || name === "expYear") {
74       const card = elements.createCardExpiryInput();
75       card.value = `${updatedData.expMonth}/${updatedData.expYear}`;
76       stripe.createToken(card).then((token) => {
77         updatedData.token = token.id;
78         setData(updatedData);
79       });
80     }
81
82     if (name === "addressLine1" || name === "addressLine2" || name === "city" || name === "state" || name === "zip") {
83       const card = elements.createCardAddressInput();
84       card.value = updatedData[name];
85       stripe.createToken(card).then((token) => {
86         updatedData.token = token.id;
87         setData(updatedData);
88       });
89     }
90
91     if (name === "email") {
92       const card = elements.createCardEmailInput();
93       card.value = updatedData[name];
94       stripe.createToken(card).then((token) => {
95         updatedData.token = token.id;
96         setData(updatedData);
97       });
98     }
99   }
100
101   const submitOrder = async () => {
102     const variables = {
103       token: data.token,
104       address: data.address,
105       city: data.city,
106       state: data.state,
107       zip: data.zip,
108       email: data.email,
109       cart: cart.items,
110     };
111
112     const response = await client.mutate({
113       mutation: gql`mutation($token: String!, $address: String!, $city: String!, $state: String!, $zip: String!, $email: String!, $cart: [CartItem!]!) {
114         charge: chargeCard(token: $token, address: $address, city: $city, state: $state, zip: $zip, email: $email, cart: $cart)
115       }
116     }, { variables };
117
118     if (response.data.charge.error) {
119       alert("There was an error processing your payment.");
120       return;
121     }
122
123     alert("Your payment was successful!");
124     resetCart();
125     setShowCart(false);
126   }
127
128   return (
129     <div>
130       <h2>Checkout</h2>
131       <form>
132         <input type="text" name="cardNumber" placeholder="Card Number" />
133         <input type="text" name="cvv" placeholder="CVV" />
134         <input type="text" name="expMonth" placeholder="Exp Month" />
135         <input type="text" name="expYear" placeholder="Exp Year" />
136         <input type="text" name="addressLine1" placeholder="Address Line 1" />
137         <input type="text" name="addressLine2" placeholder="Address Line 2" />
138         <input type="text" name="city" placeholder="City" />
139         <input type="text" name="state" placeholder="State" />
140         <input type="text" name="zip" placeholder="Zip" />
141         <input type="text" name="email" placeholder="Email" />
142         <button type="button" onClick={submitOrder}>Submit Order</button>
143       </form>
144     </div>
145   );
146 }
```

checkout.js:

```
You, 2 days ago | 1 author (You)
1 import { Elements } from "@stripe/react-stripe-js";
2 import { loadStripe } from "@stripe/stripe-js";
3 import { useInitialRender } from "@/utils/useInitial
4 import CheckoutForm from "@/components/CheckoutFor
5 import CheckoutCart from "@/components/CheckoutCart
6 const stripePromise = loadStripe("pk_test_TYooM0auv
7
8 export default function Checkout() {
9   const initialRender = useInitialRender();
10
11   if (!initialRender) return null;
12
13   return (
14     <section className="ooo">
15       <div className="ooo">
16         <div className="ooo">
17           <CheckoutCart />
18         </div>
19         <div className="ooo">
20           <Elements stripe={stripePromise}>
21             <CheckoutForm />
22           </Elements>
23         </div>
24       </div>
25     </section>
26   );
27 }
```

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

The screenshot shows the Render dashboard with two services listed: 'strapi' (PostgreSQL, PostgreSQL 15, Ohio) and 'Strapi' (Web Service, Node, Ohio). Below the dashboard is the Strapi Content Manager interface, specifically the 'dish' collection. It displays a list of 12 entries with columns for ID, NAME, DESCRIPTION, IMAGE, and STATE. Each entry has a small thumbnail image and a 'Published' status indicator.

ID	NAME	DESCRIPTION	IMAGE	STATE
1	Agnolotti	Mushroom Ragu, Fried Goat Cheese + Black...		Published
11	Chocolate Cake	Our famous cake, with a secret ingredient th...		Published
2	Classic Chicago Dog	A classic Chicago hotdog is an emblematic c...		Published
3	Deep Dish Pizza	Chicago style thick deep dish pizza with sau...		Published
4	Green Curry	Spicy and creamy Thai green curry		Published
5	Lamb with Cucumber	Succulent lamb served with fresh cucumber ...		Published
12	Mushroom Pasta	Our famous pasta with mushrooms served u...		Published
9	Pig Face	Pig Face with egg and potato straws		Published
6	Spicy Tuna Roll	Fresh tuna with a hint of spiciness wrapped i...		Published
10	Tavern Style Pizza	Chicago original tavern style pizza thin crust.		Published

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.

The screenshot shows a GraphQL playground interface with the following details:

- URL:** https://strapi-75qo.onrender.com/graphql
- PRETTIFY** and **HISTORY** buttons are visible.
- Docs** tab is selected, displaying a sidebar with a search bar and a list of available queries and mutations.
- Schema** tab is also present in the sidebar.
- Code Area:**

```

1 query {
2   restaurants {
3     data {
4       id
5       attributes {
6         name
7         dishes {
8           data {
9             attributes {
10              name
11              description
12            }
13          }
14        }
15      }
16    }
17  }
18 }
```

The code area shows a complex nested query for restaurants, including fields for id, attributes (name, dishes), and further nested attributes for dishes (name, description).

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.

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

The screenshot shows the Vercel dashboard for a project named "mit-capstone". The top section displays the deployed application's preview, status (Ready), environment (Production), and deployment duration (17s, 3h ago). Below this, the "Deployment Details" section provides a breakdown of the deployment process:

- Building:** 16s (checked)
- Deployment Summary:** All (25) files, 1s (checked)
- Next.js:** (checked)
- Static Assets:** All (25) files, HTML (7), JS (15), CSS (1), Image (1), Misc (1) (checked)
- Running Checks:** (unchecked)
- Assigning Domains:** 1s (checked)

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

The screenshot shows the Render dashboard under the 'render' account. It displays two services: 'strapi' (PostgreSQL, Available, Ohio) and 'Strapi' (Web Service, Deploy succeeded, Node, Ohio). A search bar at the top is set to 'Search services'. The 'Dashboard' tab is selected.

The screenshot shows the Render dashboard under the 'render' account. It displays a deployed 'Strapi' service (Node, Starter). The URL <https://strapi-75qo.onrender.com> is listed. The 'Logs' tab is selected, showing a log entry from August 31 at 12:14:26 PM:

```
Aug 31 12:14:26 PM [2023-08-31 17:14:26.721] http: GET /uploads/20160429_gilson_Alinea_0005_40ff37260b.jpg (5 ms) 200
```

Other tabs visible include 'Events', 'Search logs', 'Search', 'Maximize', and 'Scroll to top'. Navigation links at the bottom include 'Events', 'Logs' (selected), 'Disks', 'Environment', and 'Shell'.

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

The screenshot shows a web browser window for the application mit-capstone.vercel.app. The title bar reads "Chicago Chow". The header includes a search bar labeled "Search restaurants", a "View Cart" button with a shopping cart icon, and navigation links for "Home", "Log In", and "Sign Up". Below the header, there are five cards, each featuring a restaurant's name, a brief description, and a "View" button.

- Alinea**
A Michelin three-star restaurant with innovative dishes from Chef Grant Achatz.
[View](#)
- Dib Thai and Sushi bar**
A classic Thai restaurant with a full sushi bar known for their takeout.
[View](#)
- Girl & The Goat**
A bustling eatery by chef Stephanie Izard offering up innovative small plates & craft beers.
[View](#)
- Lou Malnati's Pizzeria**
A restaurant with red awnings and outdoor seating.
[View](#)
- Portillo's HOT DOGS**
A restaurant with a prominent sign.
[View](#)

<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.

