Plan: Building a Serverless CMS Backend on AWS

This document provides a step-by-step guide to creating a secure, scalable, and cost-effective backend for your restaurant menu CMS using AWS.

Architecture Overview

Here is a visual representation of how the services will interact:

1. **Admin User (CMS):**
   - React App -> AWS Cognito (for login)
   - React App (with auth token) -> API Gateway (secured endpoint) -> Lambda (Create/Update/Delete) -> DynamoDB
2. **Public User (Website Viewer):**
   - React App -> API Gateway (public endpoint) -> Lambda (Read-only) -> DynamoDB

Part 1: Setting up Your Database (DynamoDB)

Based on your menu structure, we'll design a table that lets you easily query for items by their main meal type (Breakfast, Lunch) and also find special featured items.

1. **Navigate to DynamoDB** in the AWS Console.
2. Click **Create table**.
3. **Table name:** RestaurantMenu
4. **Primary key:** This will now be a **composite key**.
   - **Partition key:** mealType (Type: String). This will be "Breakfast", "Lunch", "Specials", etc.
   - **Sort key:** categoryAndItemId (Type: String). This will combine the category and a unique ID to ensure every item is unique (e.g., EGGS_BENEDICT#uuid-1234).
5. **Table settings:** Choose **On-demand** for the capacity mode.
6. **Add a Global Secondary Index (GSI):** This will let us efficiently find all featured items across the entire menu.
   - Click **Add index**.
   - **Partition key:** featured (Type: String). We will add this attribute to items that are featured.
   - **Sort key:** mealType (Type: String).
   - **Index name:** FeaturedItemsIndex.
   - Leave other settings as default.
7. Click **Create table**. The table and index will take a moment to be created.

With this new structure, your menu items will be stored like this. Notice the price is now a number (which is better for calculations) and the keys match our design.

```
{
  "mealType": "Specials",
  "categoryAndItemId": "BREAKFAST_SPECIALS#uuid-5678",
  "itemId": "uuid-5678",
  "category": "BREAKFAST SPECIALS",
  "title": "Swedish Breakfast Sampler",
  "price": 16.45,
  "featured": "true"
}
```

Part 2: Creating a Secure Execution Role (IAM)

Your Lambda function will need slightly different permissions to query the new table structure efficiently.

1. **Navigate to IAM** in the AWS Console.
2. Go to **Roles** and click **Create role**.
3. **Select trusted entity:** Choose **AWS service**.
4. **Use case:** Choose **Lambda**. Click **Next**.
5. **Add permissions:**
   - Search for and select AWSLambdaBasicExecutionRole.
   - Click **Create policy**. In the new tab, click the **JSON** tab and paste the following policy, which now includes the dynamodb:Query action. **Important:** Replace <YOUR_AWS_ACCOUNT_ID> and <YOUR_AWS_REGION> with your actual account ID and region.
     ```
     {
        "Version": "2012-10-17",
        "Statement": [
           {
              "Effect": "Allow",
              "Action": [
                 "dynamodb:PutItem",
                 "dynamodb:GetItem",
                 "dynamodb:UpdateItem",
                 "dynamodb:DeleteItem",
                 "dynamodb:Scan",
                 "dynamodb:Query"
     ```

```
            ],
            "Resource": [

      "arn:aws:dynamodb:<YOUR_AWS_REGION>:<YOUR_AWS_ACCOUNT_ID>:table/
      RestaurantMenu",

      "arn:aws:dynamodb:<YOUR_AWS_REGION>:<YOUR_AWS_ACCOUNT_ID>:table/
      RestaurantMenu/index/FeaturedItemsIndex"
            ]
          }
        ]
      }
```

   - Click **Next: Tags**, then **Next: Review**.
   - **Name:** RestaurantMenuLambdaDynamoDBPolicy
   - Click **Create policy**.
   - Go back to the "Create role" browser tab. Refresh and select the RestaurantMenuLambdaDynamoDBPolicy you just created.
6. Click **Next**.
7. **Role name:** RestaurantMenuLambdaRole.
8. Click **Create role**.

Part 3: Writing the Business Logic (Lambda Functions)

Now we'll create the core logic for managing menu items. For simplicity, we'll create a single Lambda function that handles all CRUD (Create, Read, Update, Delete) operations based on the HTTP method it receives from API Gateway.

1. **Navigate to Lambda** in the AWS Console.
2. Click **Create function**.
3. Select **Author from scratch**.
4. **Function name:** manageRestaurantMenu
5. **Runtime: Node.js 18.x** (or a newer Node.js version).
6. **Architecture:** x86_64
7. **Permissions:** Expand "Change default execution role". Select **Use an existing role** and choose the RestaurantMenuLambdaRole you created in the previous step.
8. Click **Create function**.
9. In the **Code source** editor, replace the contents of index.mjs (or index.js) with the provided Node.js code for the Lambda function. **Note:** The Lambda code will also

need to be updated to work with this new, more efficient database structure.

Part 4: Setting up Authentication (Cognito)

This service will manage your admin user accounts.

1. **Navigate to Amazon Cognito** in the AWS Console.
2. Click **Create user pool**.
3. **Configure sign-in experience:**
   - Under **Cognito user pool sign-in options**, select **Email**.
4. **Configure security requirements:**
   - Leave the password policy and MFA settings as defaults for now. They are secure and suitable for this use case.
5. **Configure sign-up experience:**
   - Leave defaults. You probably don't want self-service sign-up for an admin panel, so you will create the user manually later.
6. **Configure message delivery:**
   - Choose **Send email with Cognito**. This is fine for development. For production, you may want to configure **Send email with SES**.
7. **Integrate your app:**
   - **User pool name:** RestaurantAdminPool
   - **App client name:** RestaurantCMSWebApp
   - Click **Next**.
8. Review all settings and click **Create user pool**.
9. **Create your first user:**
   - Once the pool is created, go to the **Users** tab.
   - Click **Create user**.
   - Enter your email address and a temporary password. Uncheck "Do you want to mark this phone number as verified?" and "Do you want to mark this email as verified?".
   - Click **Create user**. You will be prompted to change this password on your first login.

Part 5: Creating the Public Interface (API Gateway)

API Gateway will route incoming web requests to your Lambda function. We'll create public endpoints for reading data and secure endpoints for modifying it.

1. **Navigate to API Gateway** in the AWS Console.
2. Find the **REST API** card (not HTTP API or WebSocket) and click **Build**.
3. **Choose the protocol:** REST
4. **Create new API:** New API
5. **API name:** RestaurantMenuAPI

6. **Endpoint Type:** Regional
7. Click **Create API**.
8. **Create Authorizer:**
    - In the left navigation, click **Authorizers**.
    - Click **Create Authorizer**.
    - **Name:** CognitoAuthorizer
    - **Type:** Cognito
    - **Cognito User Pool:** Select RestaurantAdminPool.
    - **Token Source:** Authorization (This is the name of the HTTP header where the auth token will be).
    - Click **Create**.
9. **Create Resources and Methods:**
    - In the left navigation, click **Resources**.
    - Select the root / and click **Actions** -> **Create Resource**.
        - **Resource Name:** menu
        - Click **Create Resource**.
    - With the /menu resource selected, click **Actions** -> **Create Method**.
        - Select **POST**. Click the checkmark.
        - **Integration type:** Lambda Function, **Use Lambda Proxy integration**, **Lambda Function:** manageRestaurantMenu. Click **Save**.
        - In the **Method Request** card, set **Authorization** to CognitoAuthorizer.
    - Go back to the **/menu** resource. Click **Actions** -> **Create Resource**.
        - **Resource Path:** {mealType} (e.g., /menu/Breakfast)
        - Click **Create Resource**.
    - With the /{mealType} resource selected, create a **GET** method pointing to the Lambda function. This will be public.
    - With /{mealType} still selected, click **Actions** -> **Create Resource**.
        - **Resource Path:** {categoryAndItemId}
        - Click **Create Resource**.
    - With /{categoryAndItemId} selected, create **GET**, **PUT**, and **DELETE** methods, all pointing to the Lambda.
    - Secure the **PUT** and **DELETE** methods by setting their **Authorization** to CognitoAuthorizer.
10. **Enable CORS:**
    - With the /menu resource selected, click **Actions** -> **Enable CORS**.
    - Leave the defaults and click **Enable CORS and replace existing CORS headers**. This is crucial for your React app to be able to call the API.
11. **Deploy the API:**
    - Click **Actions** -> **Deploy API**.

    - **Deployment stage:** [New Stage], **Stage name:** dev
    - Click **Deploy**.
    - You will now see an **Invoke URL**. This is the base URL for your API. Save it!

Part 6: Integrating with Your React App

In your React application, the best way to interact with these AWS services is by using the **AWS Amplify** library. It simplifies authentication with Cognito and signing API requests.

1. **Install Amplify:** npm install aws-amplify
2. **Configure Amplify:** In your main index.js or App.js file, configure Amplify with the details from the services you just created. You'll need your:
    - Cognito User Pool ID
    - Cognito App Client ID
    - API Gateway Invoke URL

```
import { Amplify } from 'aws-amplify';

Amplify.configure({
 Auth: {
  Cognito: {
   userPoolId: 'YOUR_USER_POOL_ID', // e.g., us-east-1_xxxxxxxxx
   userPoolClientId: 'YOUR_APP_CLIENT_ID',
  }
 },
 API: {
  REST: {
   'RestaurantMenuAPI': {
    endpoint: 'YOUR_API_GATEWAY_INVOKE_URL', // e.g.,
https://xxxxx.execute-api.us-east-1.amazonaws.com/dev
   }
  }
 }
});
```

3. **Implement Auth and API Calls:** Use Amplify's functions to sign in, get the current user's session, and make API calls. Amplify will automatically attach the required Authorization header for your secured endpoints.
    - **Reading all Breakfast menu items (Public):**
      ```
      import { get } from 'aws-amplify/api';
      const response = await get({ apiName: 'RestaurantMenuAPI', path:
      '/menu/Breakfast' }).response;
      const items = await response.body.json();
      ```

- **Creating a new item (Secure):**
  ```
  import { post } from 'aws-amplify/api';
  // The Lambda will now need more info to build the keys
  const newItem = { mealType: "Lunch", category: "HOT SANDWICHES", title:
  "New Amazing Burger", price: 18.50 };
  const response = await post({
     apiName: 'RestaurantMenuAPI',
     path: '/menu',
     options: { body: newItem }
  }).response;
  const result = await response.body.json();
  ```

Part 7: Local Development & Testing

You asked specifically about testing in your dev environment. The gold standard for this is the **AWS Serverless Application Model (SAM) CLI**.

1. **Install AWS SAM CLI:** Follow the official AWS guide to install it.
2. **Initialize a SAM project:** In a new directory, run sam init. This will create a template template.yaml file.
3. **Define Your Stack:** Modify the template.yaml file to define all the resources you created manually above (the Lambda function, the API Gateway resources, the IAM role, etc.). This is "Infrastructure as Code" and is the best practice for managing serverless applications.
4. **Local Testing:**
   - sam build: This command builds your Lambda function.
   - sam local start-api: This command starts a local Docker container that emulates API Gateway and your Lambda function. You can then make localhost requests from your React app to this local endpoint to test your function's logic without deploying to AWS. This is perfect for rapid development and debugging.

By following this plan, you will have a professional-grade, fully serverless backend for your restaurant's CMS.