HTML, CSS, and JavaScript files to work with the provided folder structure.

**HTML (index.html):**

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Recipe Collection</title>

<link rel="stylesheet" href="styles.css">

</head>

<body>

<header>

<h1>Welcome to Recipe Collection</h1>

<p>Discover delicious recipes for every occasion!</p>

</header>

<main>

<section class="recipes">

<h2>Featured Recipes</h2>

<div class="recipe-links" id="recipe-links"></div>

</section>

</main>

<footer>

<p>&copy; 2024 Recipe Collection. All rights reserved.</p>

</footer>

<script src="scripts.js"></script>

</body>

</html>

**CSS (styles.css):**

css

/\* Add your CSS styles here \*/

body {

font-family: Arial, sans-serif;

margin: 0;

padding: 0;

}

header {

background-color: #f2f2f2;

padding: 20px;

text-align: center;

}

main {

padding: 20px;

}

footer {

background-color: #333;

color: #fff;

text-align: center;

padding: 10px;

position: fixed;

bottom: 0;

width: 100%;

}

.recipe-card {

width: 300px;

margin: 20px;

padding: 10px;

border: 1px solid #ccc;

display: inline-block;

}

.recipe-card img {

width: 100%;

height: auto;

}

.recipe-details {

margin-top: 10px;

}

.recipe-details h3 {

margin: 0;

}

.recipe-details p {

margin: 5px 0;

}

.view-recipe-button {

background-color: #4CAF50;

border: none;

color: white;

padding: 10px 20px;

text-align: center;

text-decoration: none;

display: inline-block;

font-size: 16px;

margin-top: 10px;

cursor: pointer;

}

.view-recipe-button:hover {

background-color: #45a049;

}

**JavaScript (scripts.js):**

javascript

document.addEventListener("DOMContentLoaded", function () {

// Fetch recipe data from separate text files

const recipePromises = [];

for (let i = 1; i <= 10; i++) {

recipePromises.push(fetch(`recipes/recipe${i}.txt`).then(response => response.text()));

}

Promise.all(recipePromises)

.then(recipeDataArray => {

const recipeContainer = document.getElementById("recipe-links");

recipeDataArray.forEach((recipeData, index) => {

const recipe = parseRecipeData(recipeData);

const card = document.createElement("div");

card.classList.add("recipe-card");

const image = document.createElement("img");

image.src = `images/recipe${index + 1}.jpg`;

card.appendChild(image);

const details = document.createElement("div");

details.classList.add("recipe-details");

const title = document.createElement("h3");

title.textContent = recipe.title;

details.appendChild(title);

const description = document.createElement("p");

description.textContent = recipe.description;

details.appendChild(description);

const viewButton = document.createElement("button");

viewButton.textContent = "View Recipe";

viewButton.classList.add("view-recipe-button");

viewButton.addEventListener("click", function () {

displayRecipeDetails(recipe);

});

details.appendChild(viewButton);

card.appendChild(details);

recipeContainer.appendChild(card);

});

})

.catch(error => {

console.error('Error fetching recipe data:', error);

});

function parseRecipeData(data) {

// Parse the recipe data from text format to JavaScript object

const lines = data.split('\n');

return {

title: lines[0].trim(),

description: lines.slice(1).join('\n').trim(),

// Assuming the image filename is provided in the text file

image: `recipe${lines[0].trim().toLowerCase().replace(/\s/g, '')}.jpg`

};

}

function displayRecipeDetails(recipe) {

// Implement the function to display recipe details (modal, new page, etc.)

console.log(recipe);

}

});

In this updated setup:

* The HTML file (index.html), CSS file (styles.css), and JavaScript file (scripts.js) are all stored in the same folder.
* The images are stored in the images folder, and the recipe data is stored in the recipes folder.
* JavaScript code fetches the recipe data from text files in the recipes folder and constructs recipe cards with corresponding images from the images folder.

Ensure that the filenames and paths in the JavaScript code match the actual filenames and paths in your project directory. Adjust the parsing logic if the format of your recipe text files differs.

To enable static website hosting on an S3 bucket using CloudFormation, you can add the WebsiteConfiguration property to the S3 bucket resource. Here's how you can modify the CloudFormation template to enable static website hosting:

yaml

AWSTemplateFormatVersion: '2010-09-09'

Resources:

RecipeBucket:

Type: AWS::S3::Bucket

Properties:

BucketName: mywebs3

WebsiteConfiguration:

IndexDocument: index.html

In this modified CloudFormation template:

* The WebsiteConfiguration property is added to the S3 bucket resource.
* IndexDocument specifies the name of the index document (main HTML file) for the static website. In this example, it's set to index.html, but you can change it to match the name of your main HTML file if it's different.

With this configuration, once you deploy the CloudFormation stack, the S3 bucket mywebs3 will be enabled for static website hosting, and requests to the bucket's website endpoint will serve the specified index document. Make sure to upload your HTML, CSS, JavaScript, and other static files to the bucket after deployment.

To modify the Lambda function to serve files from a container stored in Amazon Elastic Container Registry (ECR), you'll need to update the code to pull the required files from the container image. Here's how you can achieve that:

javascript

const AWS = require('aws-sdk');

const ecr = new AWS.ECR();

const docker = new AWS.Docker();

const fs = require('fs');

const path = require('path');

exports.handler = async (event) => {

try {

// Get the requested URI from the event

const { request } = event.Records[0].cf;

const { uri } = request;

// Pull the container image from ECR

const pullParams = {

imageId: {

imageTag: 'latest', // Or specify a specific tag

imageName: 'your-ecr-repo-uri'

}

};

const image = await docker.pullImage(pullParams).promise();

// Extract the requested file from the container

const containerId = image.imageId;

const extractionPath = '/tmp';

const extractionParams = {

containerId,

extractionPath

};

await docker.extractImage(extractionParams).promise();

// Read the requested file from the extraction path

const filePath = path.join(extractionPath, uri.substr(1)); // Remove the leading '/' from URI

const fileContent = fs.readFileSync(filePath, 'utf-8');

// Return the file content as the response

return {

status: '200',

statusDescription: 'OK',

headers: {

'content-type': [{ key: 'Content-Type', value: 'text/html' }]

},

body: fileContent

};

} catch (error) {

console.error('Error serving file from container:', error);

return {

status: '500',

statusDescription: 'Internal Server Error',

body: 'Internal Server Error'

};

}

};

In this modified Lambda function:

1. We use the AWS SDK to interact with Amazon ECR and AWS Docker APIs.
2. We pull the latest container image from the specified ECR repository.
3. We extract the requested file from the pulled container image.
4. We read the content of the requested file and return it as the response.

Please ensure that your Lambda function has appropriate IAM permissions to access ECR and Docker APIs. Additionally, update 'your-ecr-repo-uri' with the URI of your ECR repository. This Lambda function assumes that the requested files are present in the root directory of the container image. Adjust the extraction logic if the files are located in a different directory within the container.

To accomplish the task of containerizing the site components, deploying them to Amazon EKS, and managing the EKS resources using AWS CloudFormation, we'll go through each step:

**1. Dockerize Site Components:**

Create Dockerfiles for each component (index.html, styles.css, script.js, images folder, recipes folder).

Here's a basic example of Dockerfile for serving static files:

Dockerfile

# Dockerfile for serving static files

# Use nginx as the base image

FROM nginx:alpine

# Copy HTML, CSS, JavaScript, images, and recipes to the appropriate directories in the nginx server

COPY index.html /usr/share/nginx/html/

COPY styles.css /usr/share/nginx/html/

COPY script.js /usr/share/nginx/html/

COPY images/ /usr/share/nginx/html/images/

COPY recipes/ /usr/share/nginx/html/recipes/

Ensure that you have all necessary files in the same directory as the Dockerfile before building the Docker image.

**2. Create Kubernetes Deployment and Service YAML Files:**

Here's an example of Kubernetes Deployment and Service YAML files:

yaml

# deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: myweb-deployment

spec:

replicas: 2

selector:

matchLabels:

app: myweb

template:

metadata:

labels:

app: myweb

spec:

containers:

- name: myweb-container

image: <ECR-repo-uri>:<tag>

ports:

- containerPort: 80

yaml

# service.yaml

apiVersion: v1

kind: Service

metadata:

name: myweb-service

spec:

selector:

app: myweb

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

Replace <ECR-repo-uri> and <tag> with the appropriate values for your Docker image repository URI and tag.

**3. Use AWS CloudFormation for Amazon EKS:**

AWS CloudFormation doesn't directly support creating Amazon EKS clusters. However, you can use CloudFormation to create and manage the resources required for EKS, such as the worker nodes and associated resources like VPC, subnets, IAM roles, etc.

Here's an example CloudFormation template for creating a worker node group for Amazon EKS:

yaml

# eks-worker-node.yaml

Resources:

MyEKSNodeGroup:

Type: AWS::EKS::Nodegroup

Properties:

ClusterName: !Ref EKSCluster

NodegroupName: my-eks-nodegroup

Subnets: [subnet-1a, subnet-1b] # Specify your subnets

NodeRole: arn:aws:iam::123456789012:role/NodeInstanceRole # Specify your NodeInstanceRole ARN

ScalingConfig:

MinSize: 1

MaxSize: 3

DesiredSize: 2

Tags:

- Key: Name

Value: my-eks-nodegroup

This CloudFormation template creates a worker node group for the specified Amazon EKS cluster, specifying the subnets, IAM role for nodes, and scaling configuration.

You'll need to manually create the EKS cluster itself using the AWS Management Console or AWS CLI, as CloudFormation does not directly support creating EKS clusters at this time.

After creating the EKS cluster, you can deploy your Kubernetes Deployment and Service YAML files using kubectl or any Kubernetes management tool. Make sure you have the necessary IAM permissions to interact with the EKS cluster.