

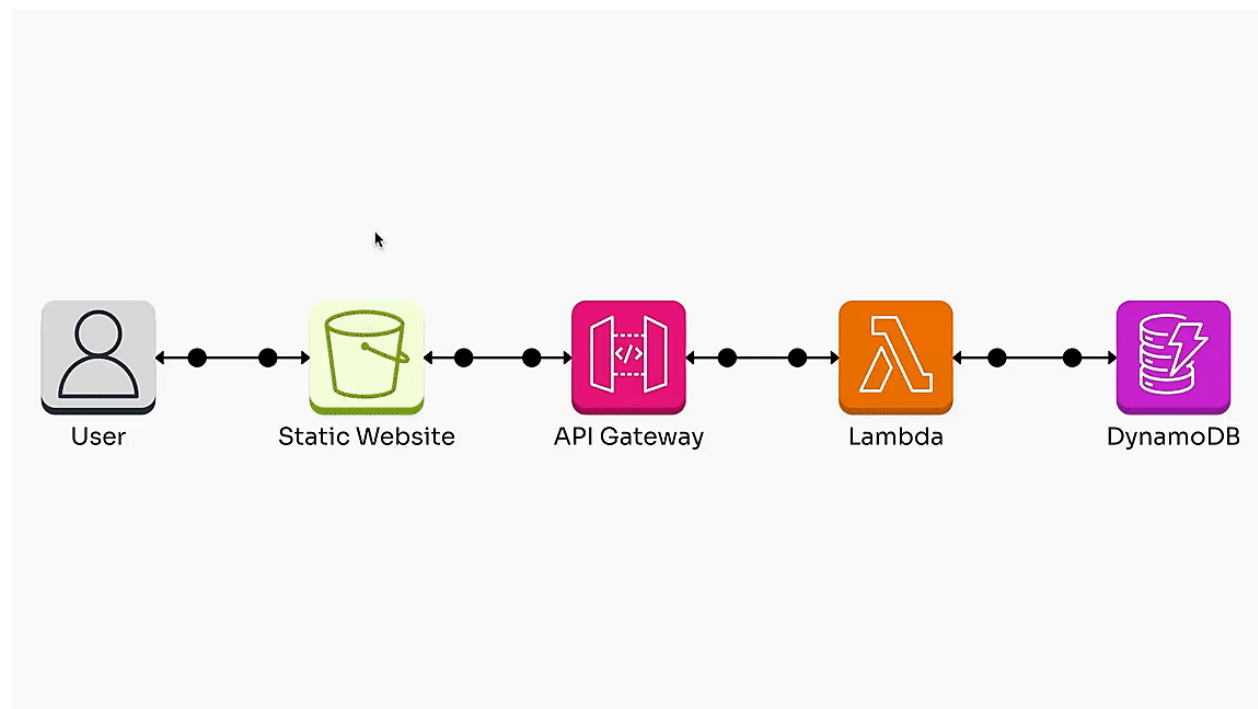
# Project Title: Serverless Web Application using AWS

## Project Overview

The **Serverless Application** is a cloud-based web application designed to store, retrieve, and manage recipes. The application is built using **AWS serverless architecture**, ensuring scalability, high availability, and cost-efficiency. The frontend is hosted on **Amazon S3** and interacts with backend services like **API Gateway**, **AWS Lambda**, and **DynamoDB**.

---

## Architecture



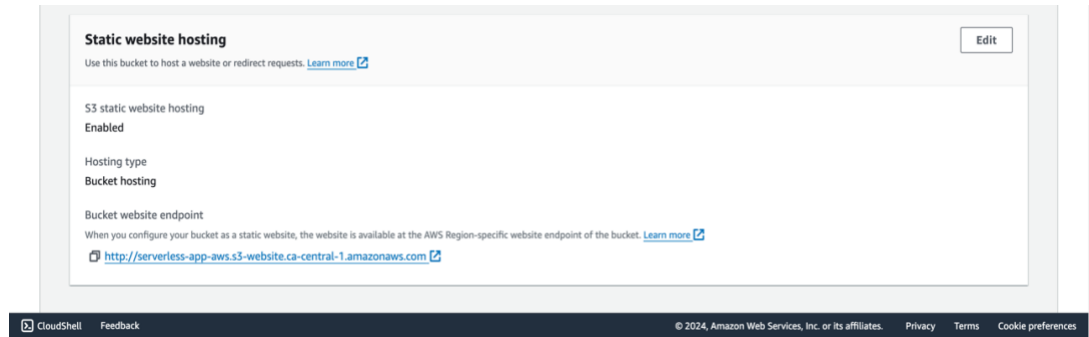
The application follows a fully serverless design:

- **Frontend:** Hosted on **Amazon S3** as a static website using HTML, CSS, and JavaScript.
- **API Gateway:** Provides a RESTful API to route requests from the frontend to backend services.
- **AWS Lambda:** Handles backend logic, fetching the data from DynamoDB.
- **DynamoDB:** Stores the recipe data in a NoSQL table.

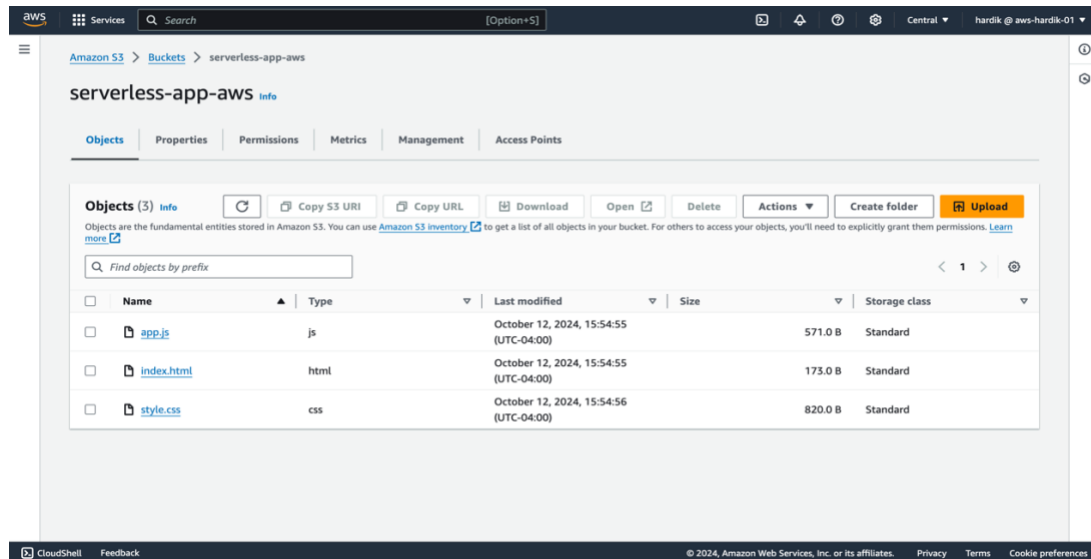
## Detailed Workflow

### 1. Frontend (Static Website on S3):

- The application's user interface is hosted in an Amazon S3 bucket (serverless-app-aws) with static website hosting enabled.



- The S3 bucket contains index.html, style.css, and app.js, allowing users to interact with the application through their web browsers.

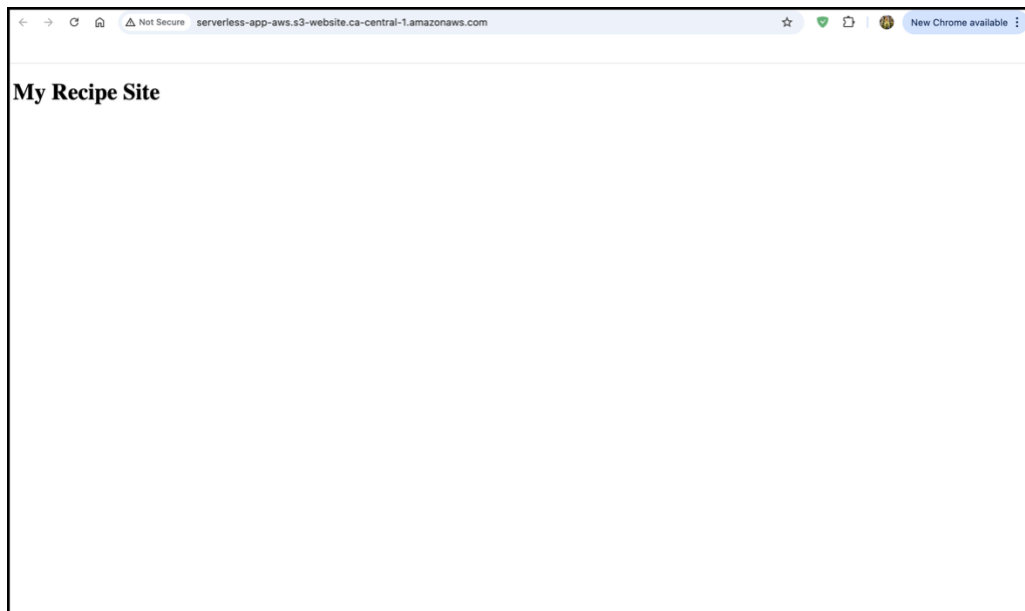


- Attached a read-only policy to the S3 bucket.

Policy

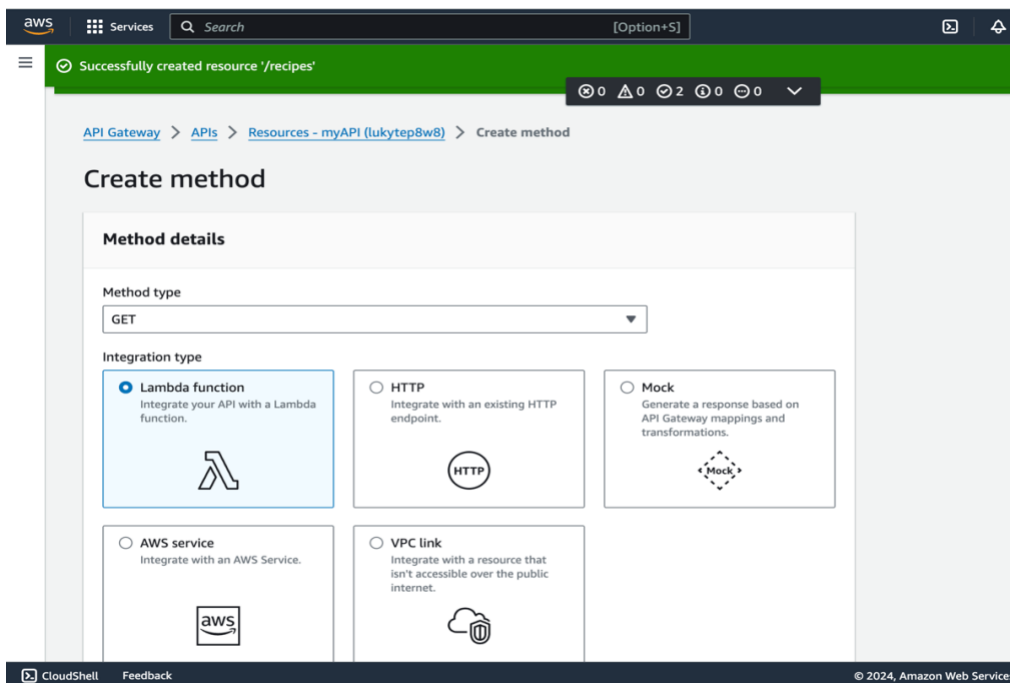
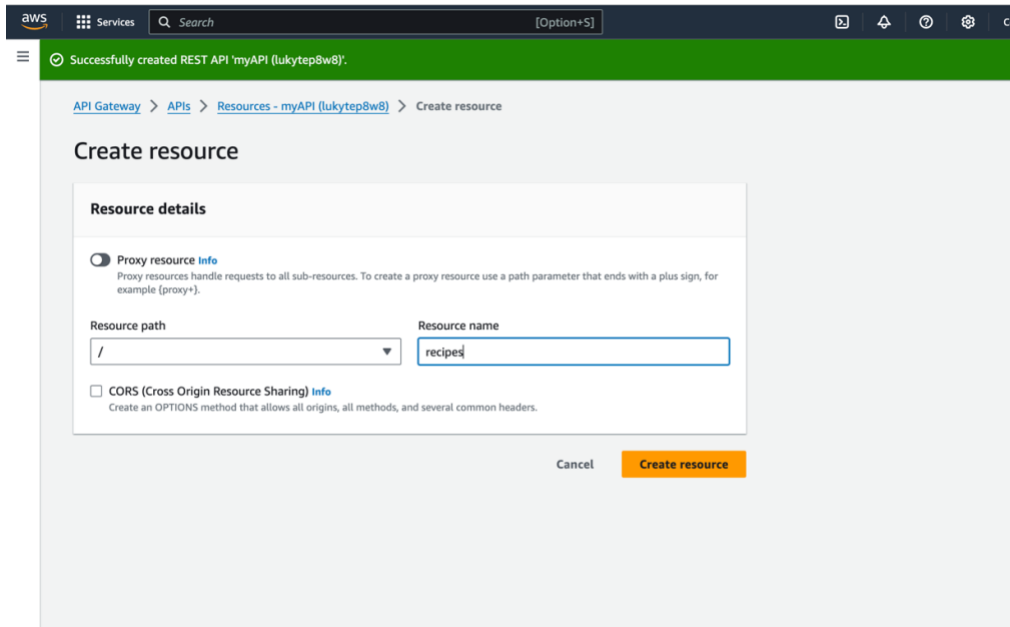
```
1 {  
2   "Version": "2012-10-17",  
3   "Statement": [  
4     {  
5       "Sid": "PublicReadGetObject",  
6       "Effect": "Allow",  
7       "Principal": "*",  
8       "Action": "s3:GetObject",  
9       "Resource": "arn:aws:s3:::serverless-app-aws/*"  
10    }  
11  ]  
12 }
```

- Accessed a website before implementing backend services.

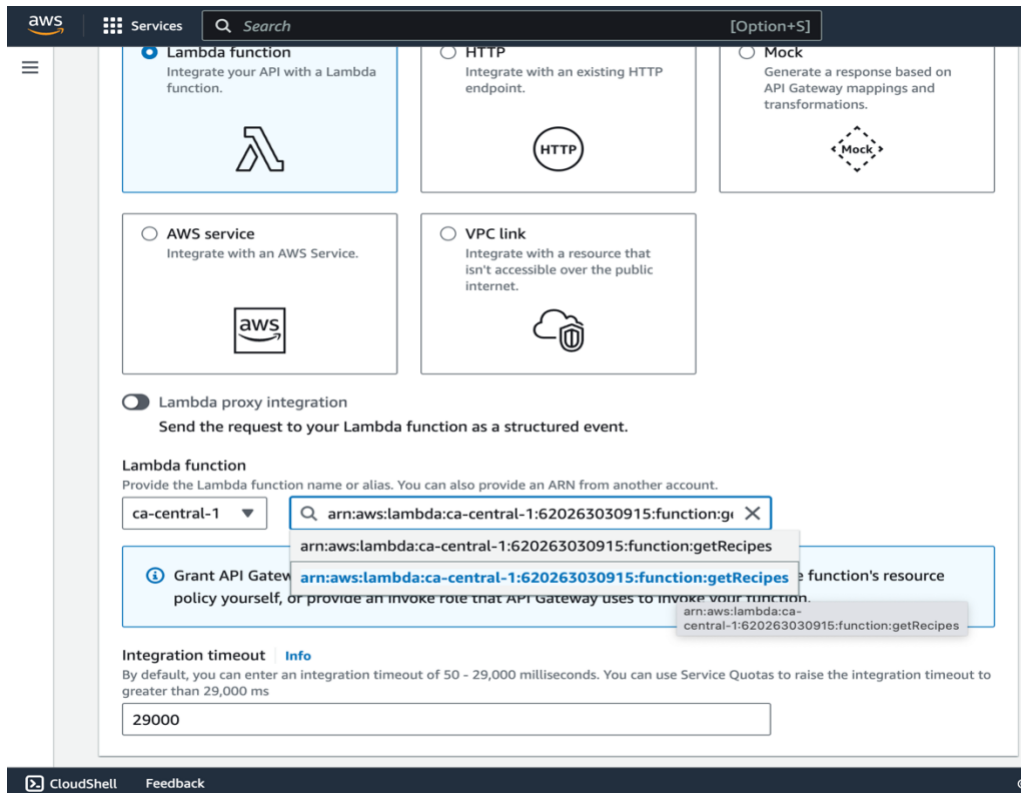


## 2. API Gateway:

- Amazon API Gateway manages and processes incoming requests to the application's backend.
- A REST API (myAPI) is created with a resource /recipes and a GET method attached to retrieve recipe data.



- The API Gateway forwards these requests to a Lambda function and sends responses back to the frontend.



aws Services Search [Option+S]

**Lambda function**  
Integrate your API with a Lambda function.

**HTTP**  
Integrate with an existing HTTP endpoint.

**Mock**  
Generate a response based on API Gateway mappings and transformations.

**AWS service**  
Integrate with an AWS Service.

**VPC link**  
Integrate with a resource that isn't accessible over the public internet.

**Lambda proxy integration**  
Send the request to your Lambda function as a structured event.

**Lambda function**  
Provide the Lambda function name or alias. You can also provide an ARN from another account.

ca-central-1

arn:aws:lambda:ca-central-1:620263030915:function:getRecipes

arn:aws:lambda:ca-central-1:620263030915:function:getRecipes

arn:aws:lambda:ca-central-1:620263030915:function:getRecipes

arn:aws:lambda:ca-central-1:620263030915:function:getRecipes

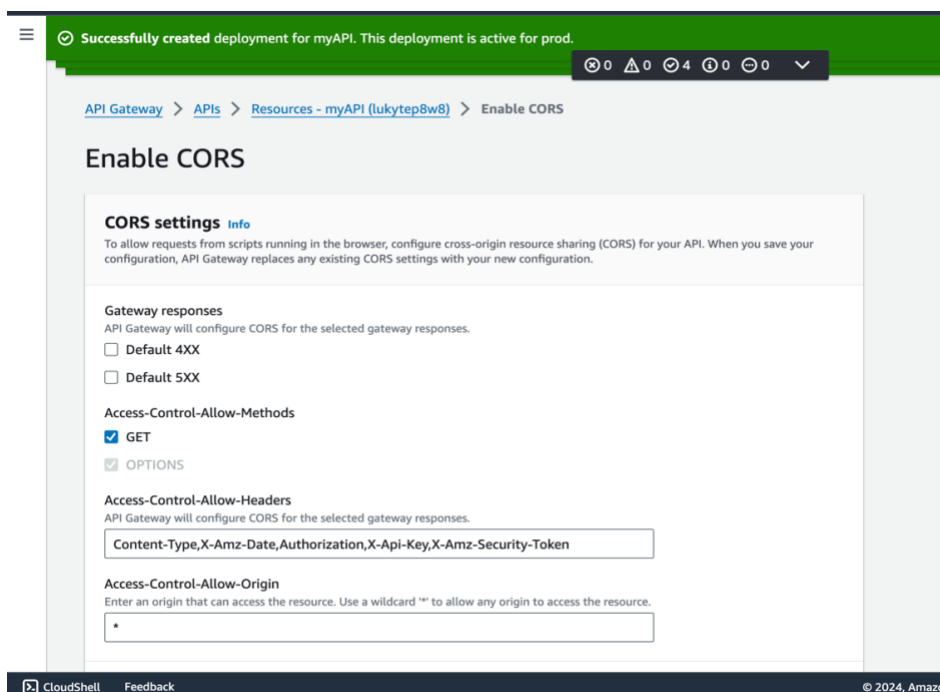
**Grant API Gateway**  
policy yourself, or provide an invoke role that API Gateway uses to invoke your function.

**Integration timeout** Info  
By default, you can enter an integration timeout of 50 - 29,000 milliseconds. You can use Service Quotas to raise the integration timeout to greater than 29,000 ms

29000

CloudShell Feedback

- Enabled the CORS to cross-origin requests securely.



Successfully created deployment for myAPI. This deployment is active for prod.

API Gateway > APIs > Resources - myAPI (lukytp8w8) > Enable CORS

## Enable CORS

**CORS settings** info  
To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API. When you save your configuration, API Gateway replaces any existing CORS settings with your new configuration.

**Gateway responses**  
API Gateway will configure CORS for the selected gateway responses.

☐ Default 4XX

☐ Default 5XX

**Access-Control-Allow-Methods**

☒ GET

☒ OPTIONS

**Access-Control-Allow-Headers**  
API Gateway will configure CORS for the selected gateway responses.

Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token

**Access-Control-Allow-Origin**  
Enter an origin that can access the resource. Use a wildcard "\*" to allow any origin to access the resource.

\*

CloudShell Feedback

© 2024, Amazon

- Deployed an API creating new stage called “prod” and copied the URL to modify the app.js file.

### Deploy API

Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. [Learn more](#)

⚠ When you deploy an API to an existing stage, you immediately overwrite the current stage configuration with a new active deployment.

Stage

prod

Deployment description

Cancel Deploy

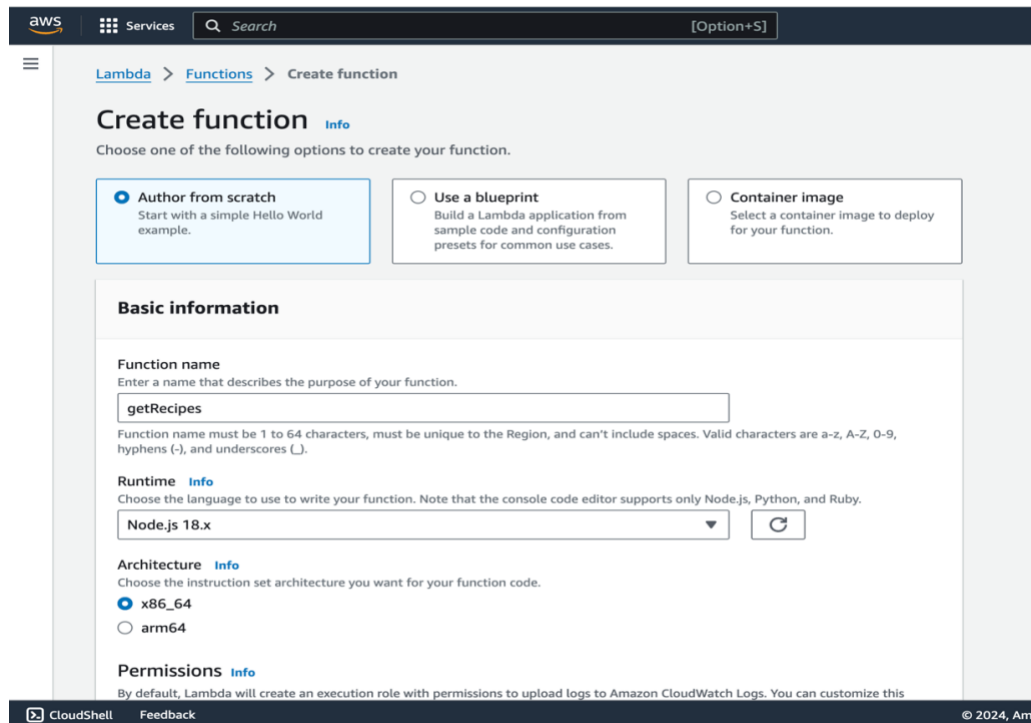
✓ Copied

https://luckytep8w8.execute-api.ca-central-1.amazonaws.com/prod

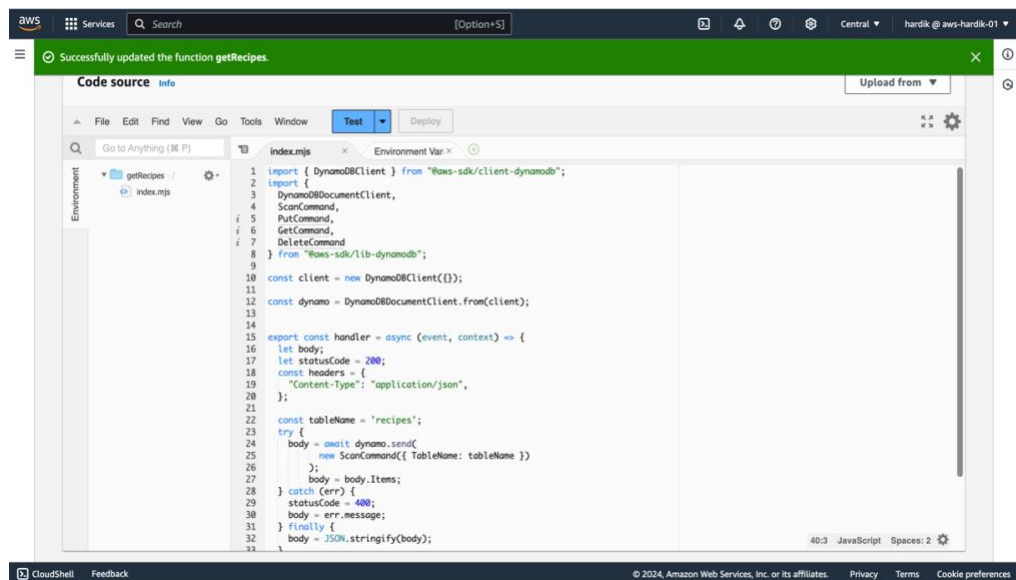
```
index.html  # styles.css  JS script.js  JS app.js
Users > hardikrathod > Downloads > Ex_Files_Building_Serverless_Applications_AWS > Exercise Files > JS app.js > ...
1  window.onload = function() {
2      fetch('https://luckytep8w8.execute-api.ca-central-1.amazonaws.com/prod/recipes')
3      .then(response => response.json())
4      .then(data => {
5          const recipeList = document.getElementById('recipeList');
6          const recipes = JSON.parse(data.body);
7          recipes.forEach(recipe => {
8              const recipeItem = document.createElement('div');
9              recipeItem.textContent = `${recipe.name}: ${recipe.instructions}`;
10             recipeList.appendChild(recipeItem);
11         });
12     });
13 };
14
```

### 3. Lambda Function (getRecipes):

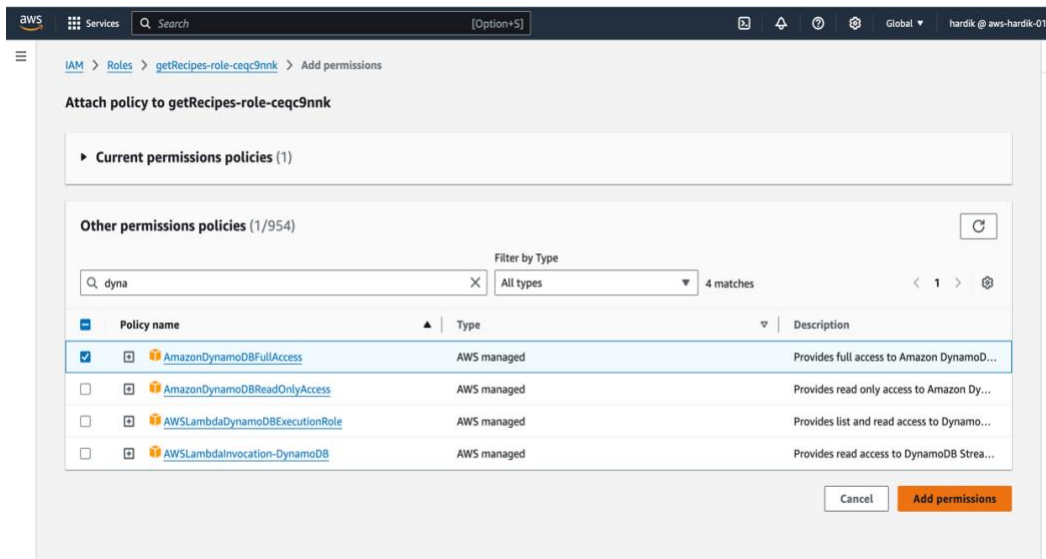
- A Node.js-based Lambda function (getRecipes) retrieves data from the DynamoDB table and returns it to the API Gateway.
- Created a Lambda function (getRecipes) using Node.js to fetch recipes from DynamoDB.



- The function uses the AWS SDK (@aws-sdk/lib-dynamodb) to interact with DynamoDB.



- It executes a ScanCommand to retrieve all items (recipes) from the recipes table and sends the data back as a JSON response.
- The function has full access to DynamoDB and is responsible for retrieving the list of recipes.





#### 4. DynamoDB (Data Storage):

- DynamoDB stores the recipe data in a table named recipes.

The screenshot shows the 'Create table' page in the AWS Management Console. The page is titled 'Create table' and has a breadcrumb trail: 'DynamoDB > Tables > Create table'. Below the title, there is a 'Table details' section with an 'info' icon. It states: 'DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.' The 'Table name' field is labeled 'This will be used to identify your table.' and contains the text 'recipes'. Below this, it says 'Between 3 and 255 characters, containing only letters, numbers, underscores (\_), hyphens (-), and periods (.)'. The 'Partition key' section is labeled 'The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.' The 'id' field is selected as the partition key, and the data type is 'String'. Below this, it says '1 to 255 characters and case sensitive.' The 'Sort key - optional' section is labeled 'You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.' The 'Enter the sort key name' field is empty, and the data type is 'String'. Below this, it says '1 to 255 characters and case sensitive.' The 'Table settings' section is partially visible at the bottom.

- The Lambda function reads from this table whenever the frontend requests recipe data.
- Each recipe is stored as an item with attributes like id, name, ingredients, and instructions.

The screenshot shows the 'Create item' page in the AWS Management Console. The page is titled 'Create item' and has a breadcrumb trail: 'DynamoDB > Explore Items: recipes > Create item'. Below the title, there is a 'Form' button and a 'JSON view' button. The 'Attributes' section is active, and it shows a JSON representation of a recipe item. The JSON is as follows:

```
1 {
2   "id": {"S": "1"},
3   "name": {"S": "Pasta"},
4   "ingredients": {"S": "Pasta, Tomatoes, Olive Oil, Garlic, Salt, Pepper"},
5   "instructions": {"S": "Boil the pasta. Saute garlic in olive oil. Add tomatoes, cooked pasta. Season with salt and pepper. Serve hot."}
6 }
```

- Accessed a website after implementing backend services.



## Technologies Used

- **AWS Services:**
  - Amazon S3 (Static Website Hosting)
  - AWS Lambda (Serverless Backend)
  - Amazon API Gateway (REST API)
  - Amazon DynamoDB (NoSQL Database)
- **Frontend Technologies:**
  - HTML, CSS, JavaScript

## How the Application Works

1. **Accessing the Application:** The frontend hosted on S3 provides an easy-to-use interface where users can view recipe data.
2. **Sending Requests:** The JavaScript code in app.js sends a **GET request** to the API Gateway to retrieve the list of recipes from DynamoDB.
3. **API Gateway:** API Gateway forwards the request to the **Lambda function**.
4. **Lambda Processing:** The Lambda function executes the necessary logic to fetch all recipe data from the DynamoDB table using a **ScanCommand**.
5. **Response Handling:** The Lambda function sends the data back to the API Gateway, which forwards it to the frontend.
6. **Displaying Data:** The frontend receives the response and dynamically displays the list of recipes on the website.

## Challenges Faced

- **CORS Configuration:** Handling CORS (Cross-Origin Resource Sharing) to allow communication between the frontend (hosted on S3) and API Gateway required specific configurations to avoid blocking requests.
- **Lambda and DynamoDB Integration:** Ensuring smooth data retrieval from DynamoDB through Lambda while managing permissions and handling errors efficiently was key to the project's success.

## Skills Highlighted

- **AWS Cloud:** Serverless architecture using Lambda, API Gateway, DynamoDB, and S3.
- **Full-Stack Development:** Experience in integrating backend services with a dynamic frontend.
- **JavaScript:** Use of modern JavaScript to interact with REST APIs and dynamically update the frontend.
- **NoSQL Databases:** Practical experience in using DynamoDB for scalable and efficient data storage.

## Conclusion

This project highlights the power of AWS Serverless Architecture in building scalable, cost-efficient web applications. By leveraging services like S3, API Gateway, Lambda, and DynamoDB, I was able to create an efficient, highly available recipe application with no need for traditional server management.