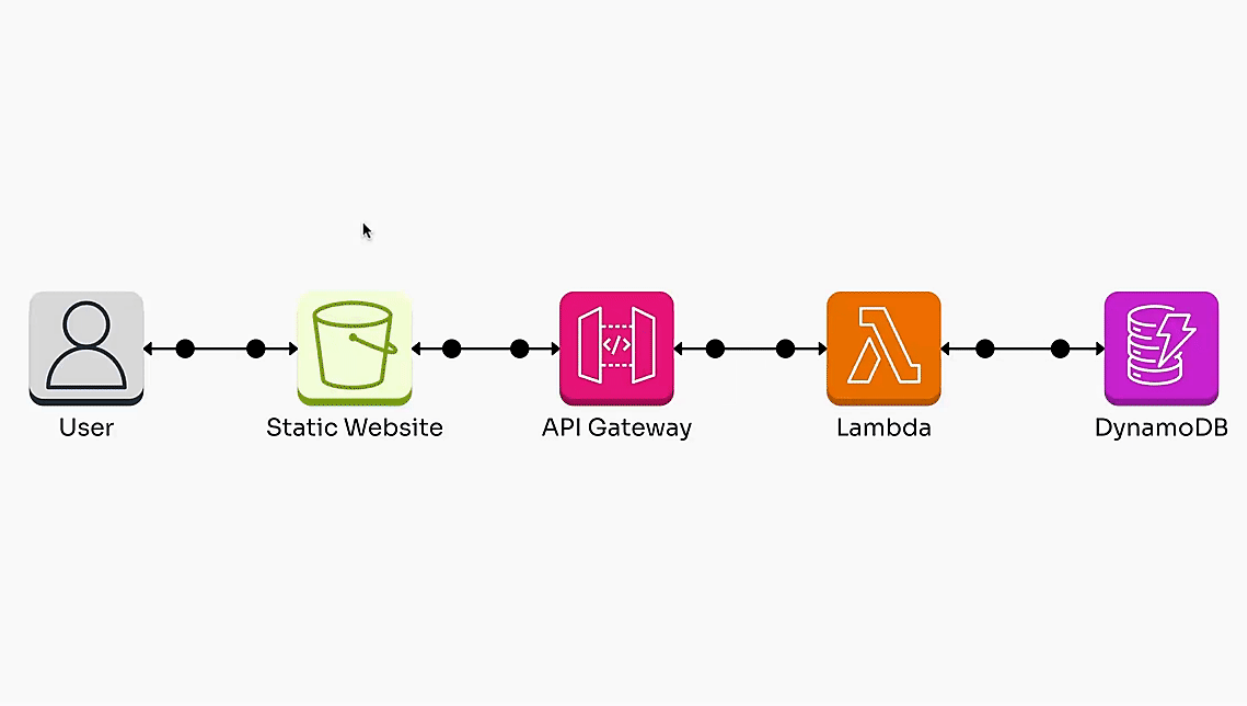
Project Title: Serverless Web Application using AWS

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



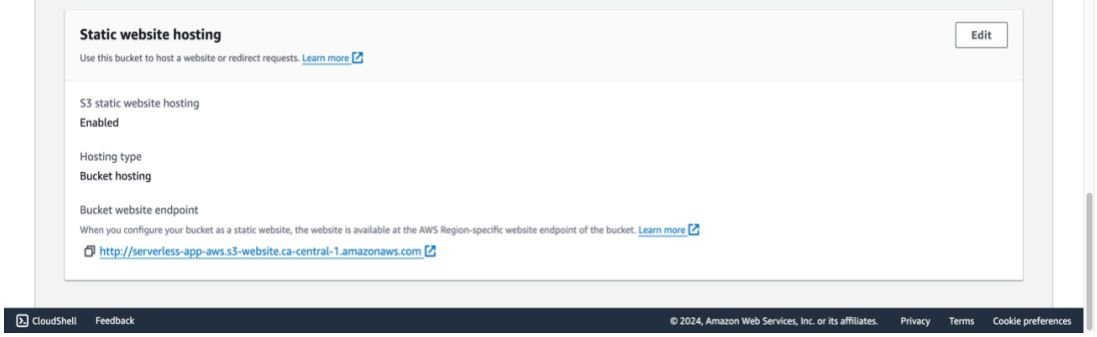
The application follows a fully serverless design:

- **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.

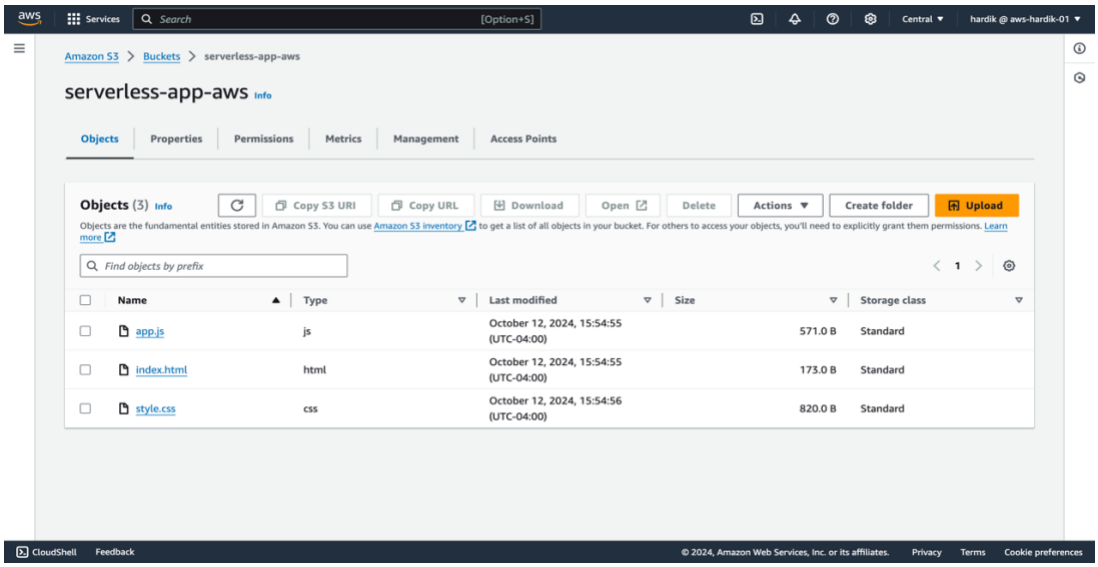
Detailed Workflow

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.

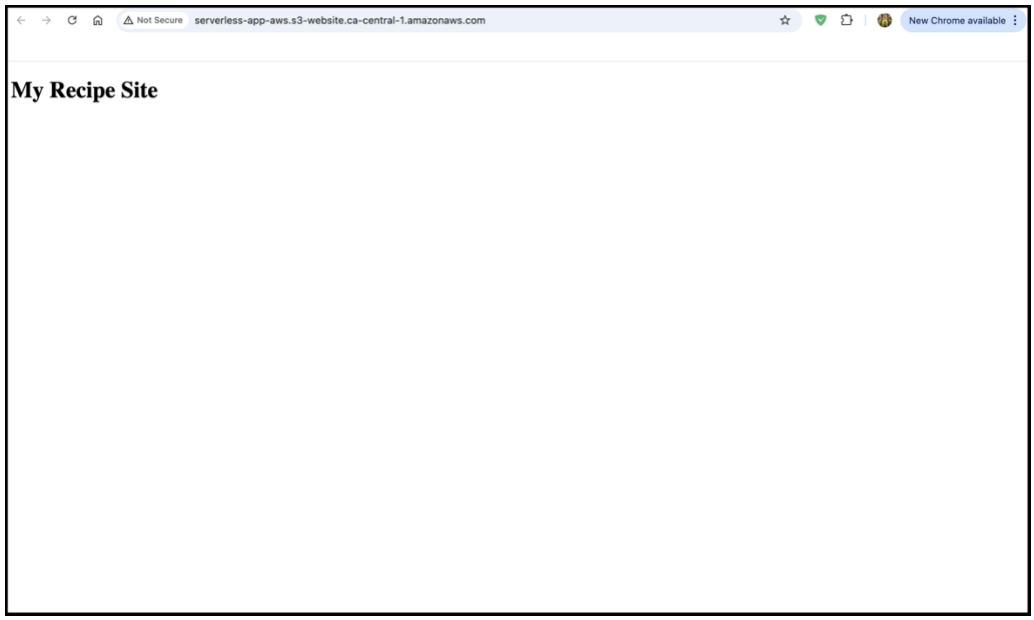


- Attached a read-only policy to the S3 bucket.

Policy

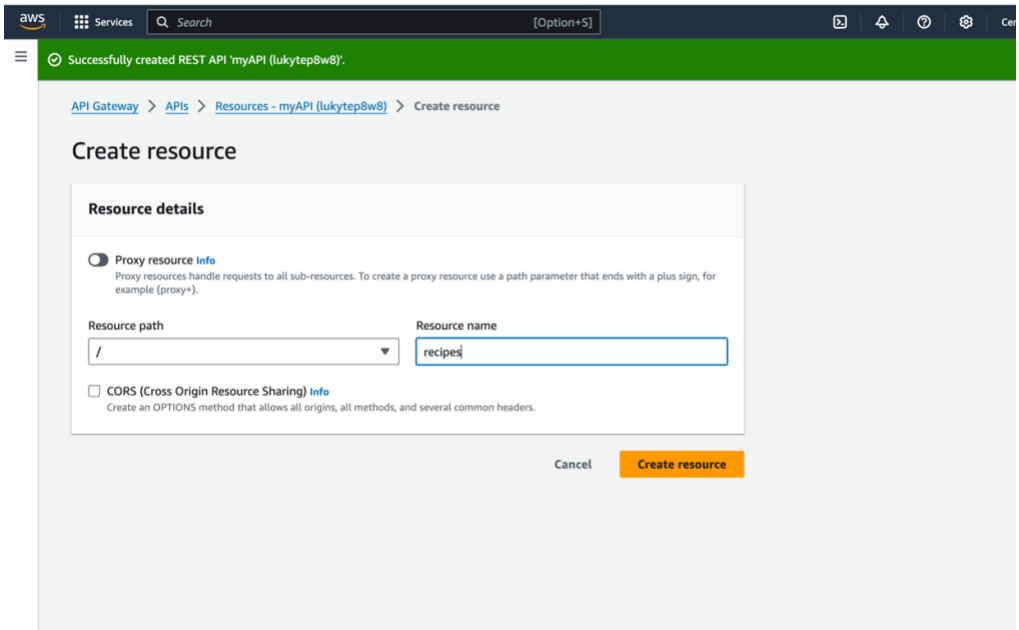
```
1  {
2    "Version": "2012-10-17",
3    "Statement": [
4      {
5        "Sid": "PublicReadGetObject",
6        "Effect": "Allow",
7        "Principal": "*",
8        "Action": "s3:GetObject",
9        "Resource": "arn:aws:s3:::serverless-app-aws/*"
10     }
11   ]
12 }
```

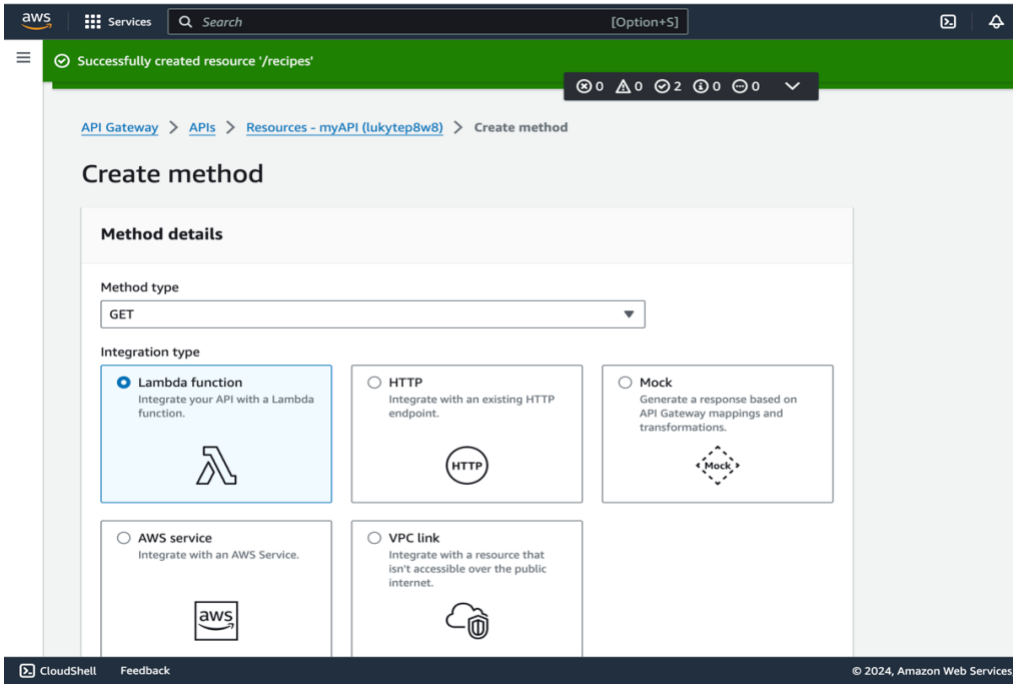
- Accessed a website before implementing backend services.



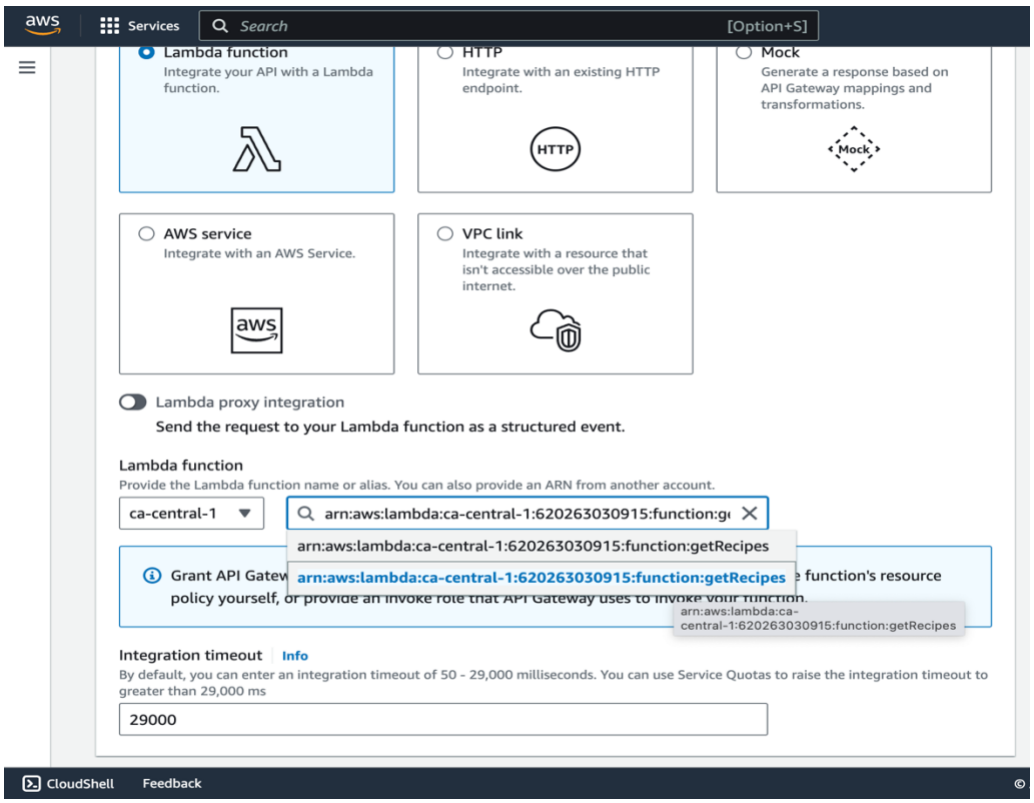
2. **API Gateway**:

- 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.

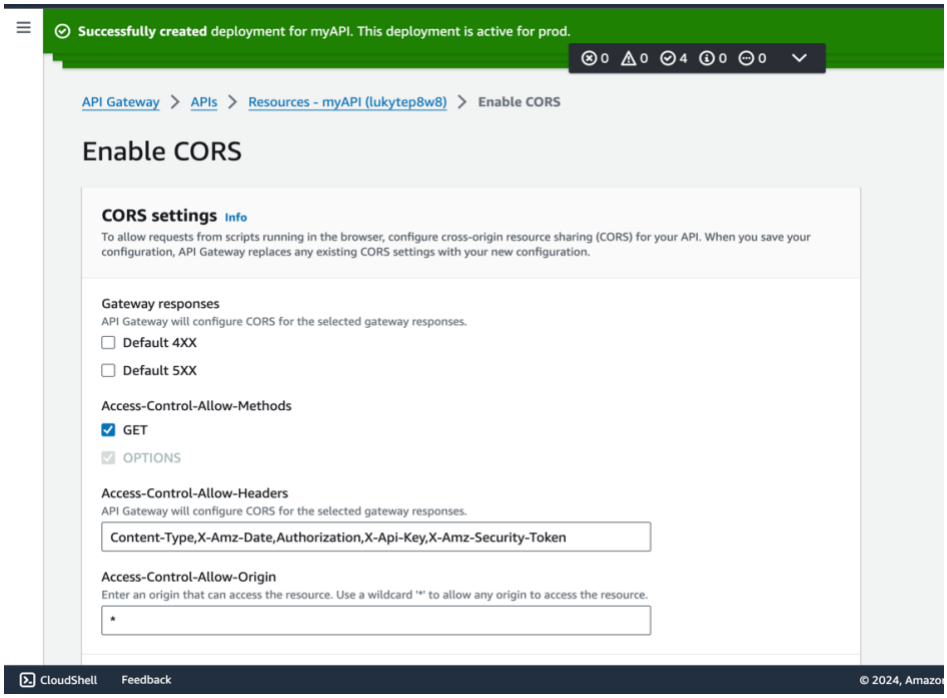




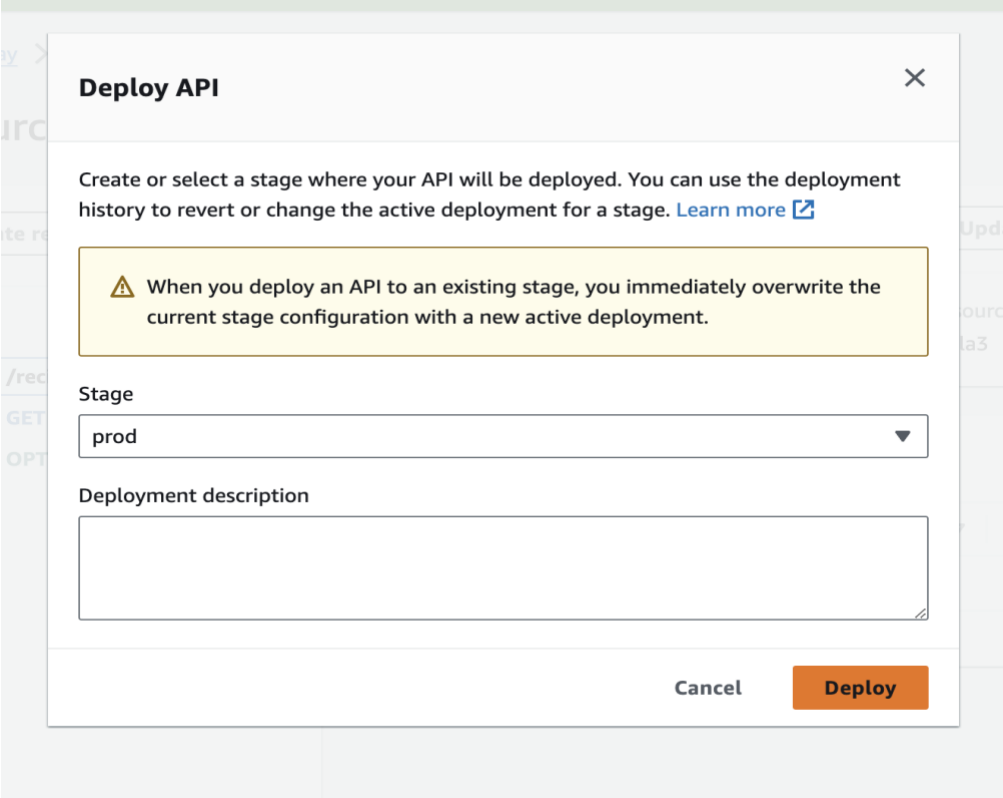
- The API Gateway forwards these requests to a Lambda function and sends responses back to the frontend.

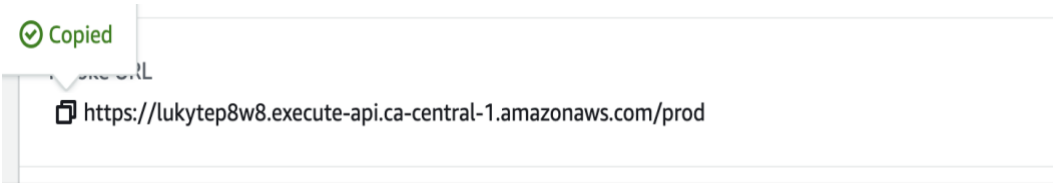


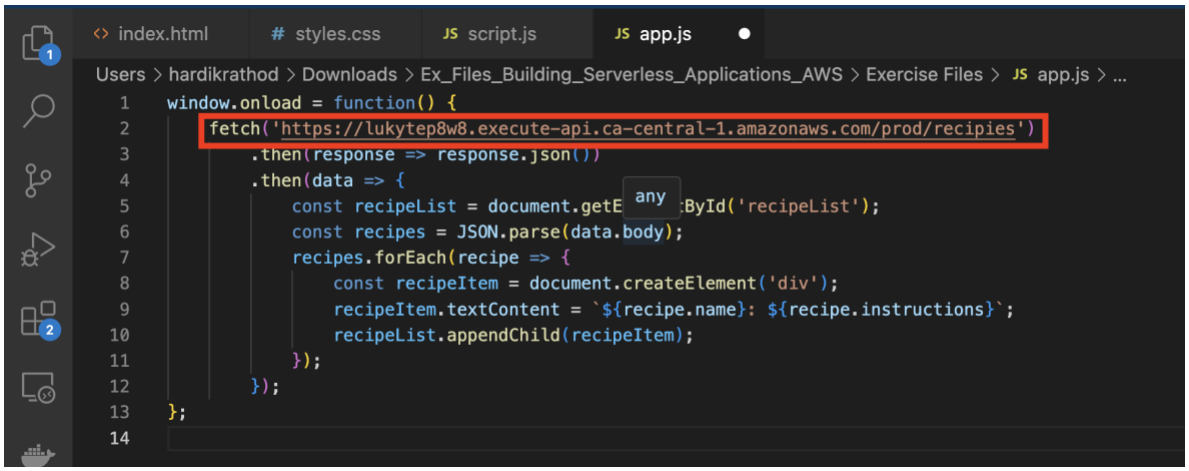
- Enabled the CORS to cross-origin requests securely.



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

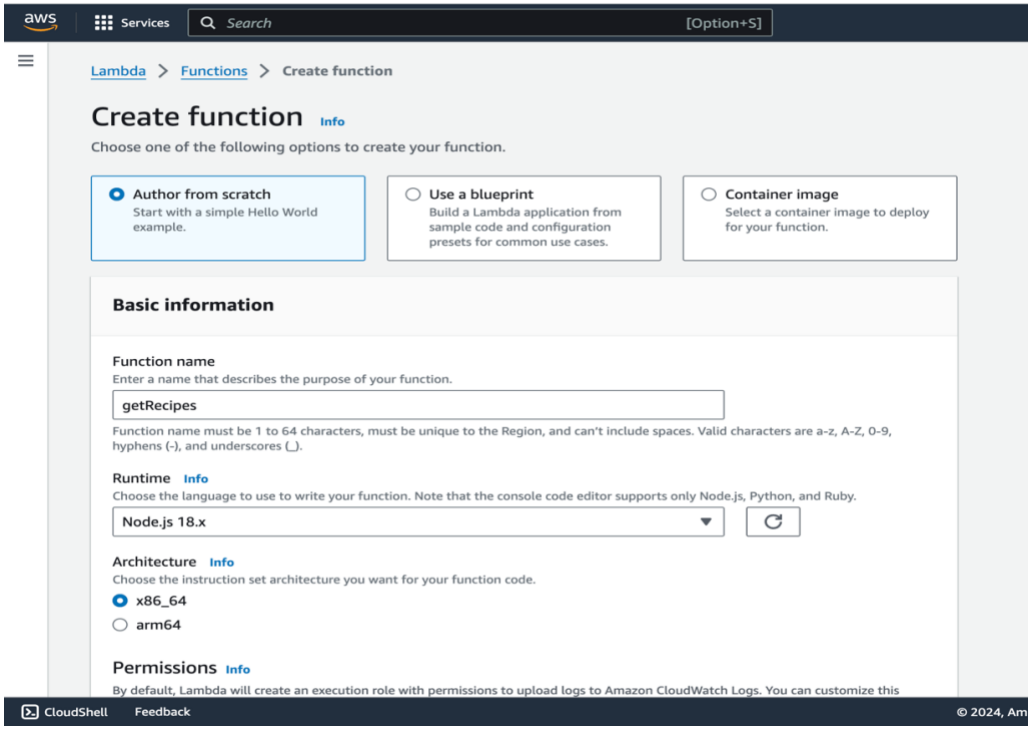




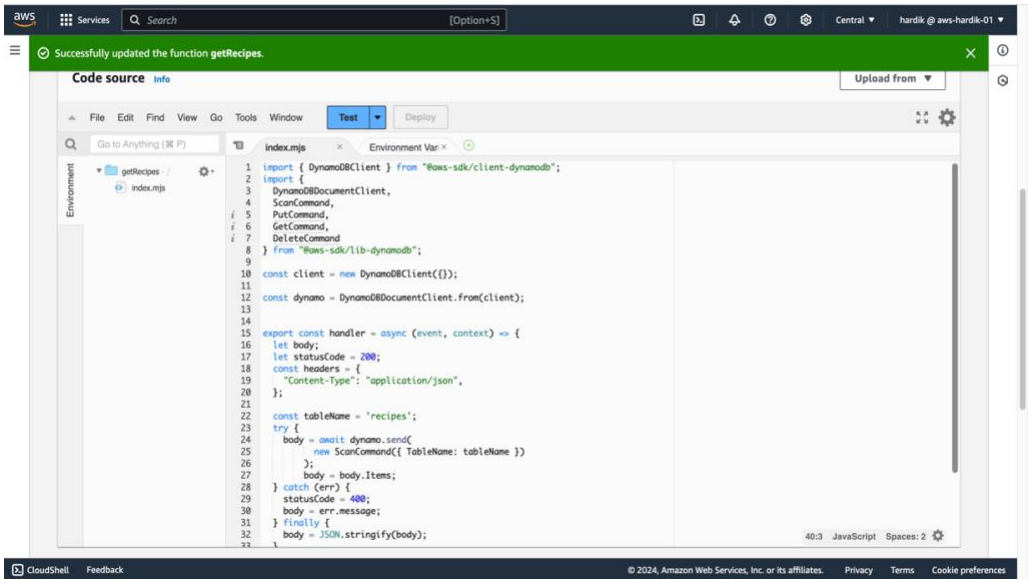


3. **Lambda Function (getRecipes)**:

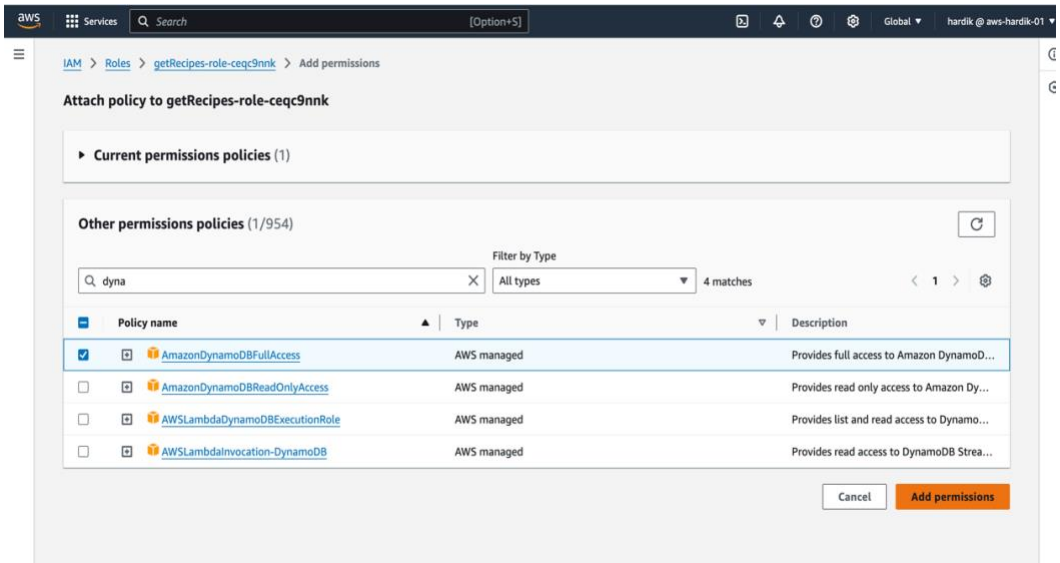
- A Node.js-based Lambda function (getRecipes) retrieves data from the DynamoDB table and returns it to the API Gateway.
- Created a Lambda function (getRecipes) using Node.js to fetch recipes from DynamoDB.



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

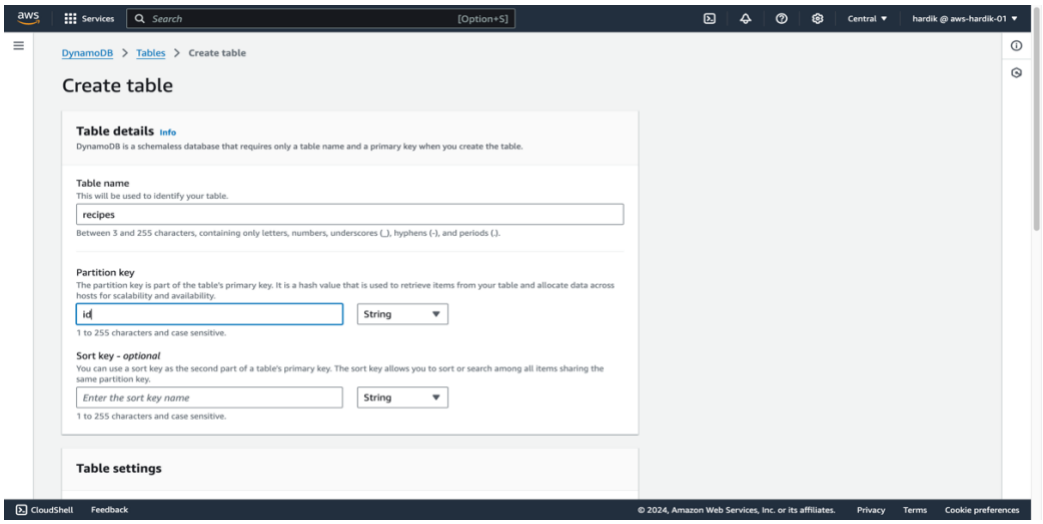


- It executes a ScanCommand to retrieve all items (recipes) from the recipes table and sends the data back as a JSON response.
- The function has full access to DynamoDB and is responsible for retrieving the list of recipes.

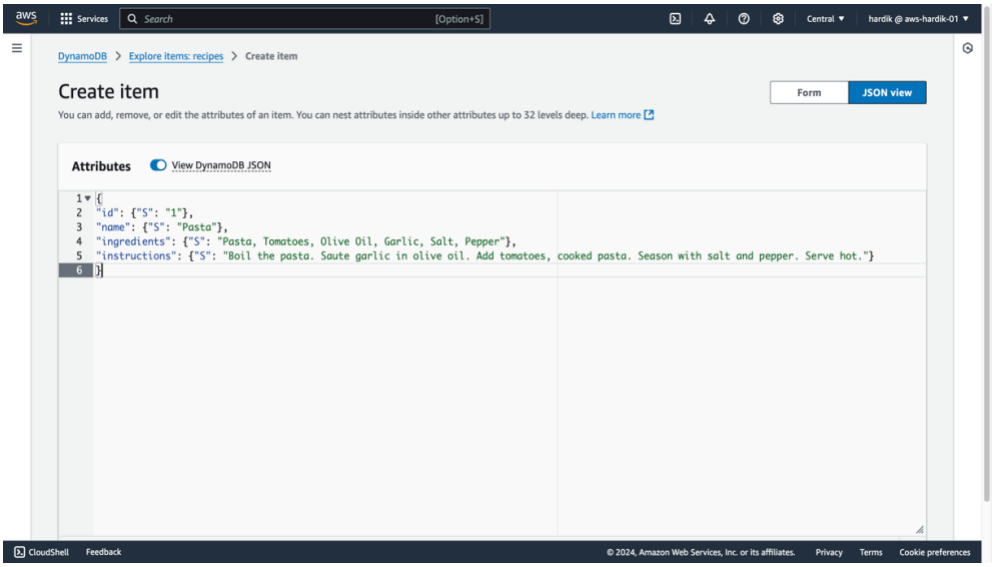


4. **DynamoDB (Data Storage)**:

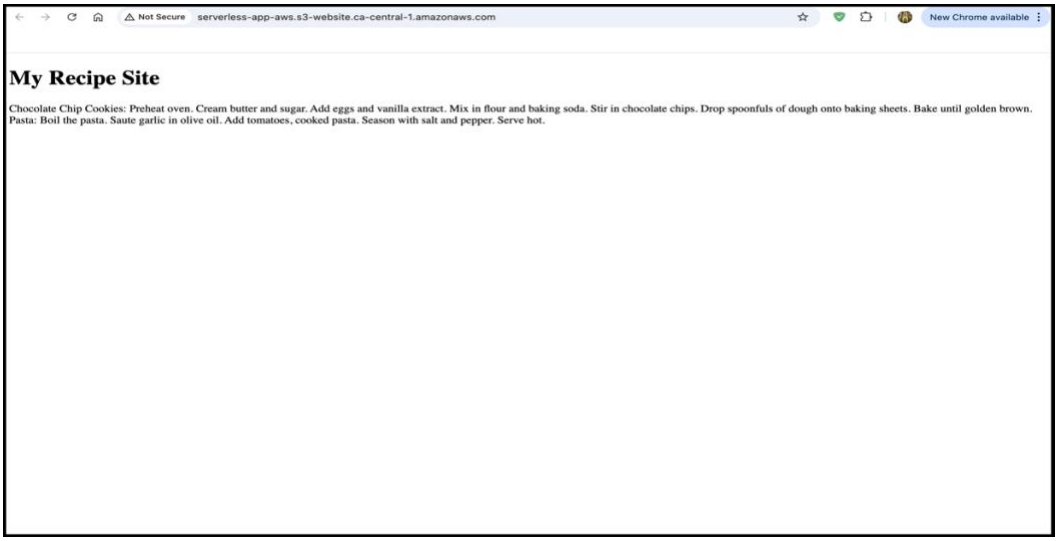
- DynamoDB stores the recipe data in a table named recipes.



- The 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.



- 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.