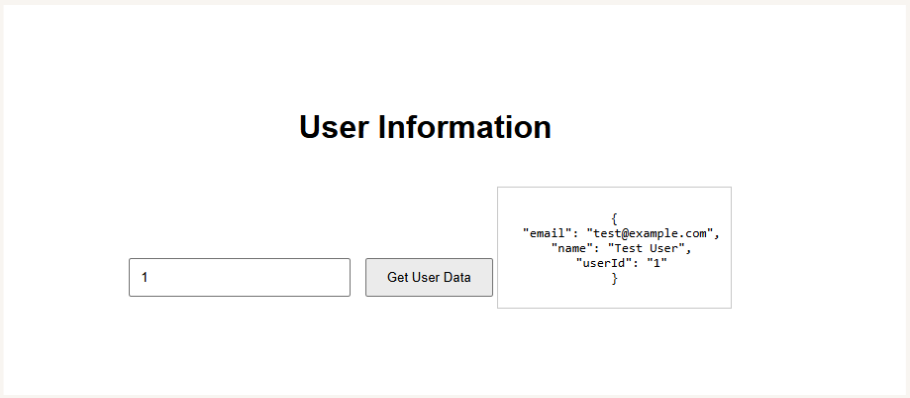
nextwork.org

Build a Three-Tier Web App

Louis Moyo



Louis Moyo
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to build a full three-tier architecture using AWS services. I'm doing this project to learn how to connect the presentation, logic, and data tiers together by using S3, CloudFront, Lambda, API Gateway, and DynamoDB to create a scalable, serverless web application.

Tools and concepts

Services I used were Amazon S3, CloudFront, API Gateway, DynamoDB, and Lambda functions. Key concepts I learnt include how CloudFront speeds up global content delivery, how browsers enforce security with CORS, how to configure API Gateway to accept cross-origin requests, and how to return proper CORS headers directly in Lambda responses so that a frontend and backend can talk securely.

Project reflection

This project took me approximately 3 hours across multiple troubleshooting steps. The most challenging part was understanding why my CloudFront site could not talk to my API due to CORS errors and realising that API Gateway configuration alone wasn't enough. It was most rewarding to see the final connection work after updating my Lambda function to return the correct headers, proving I had built a fully functional end-to-end system.

L

Louis Moyo
NextWork Student

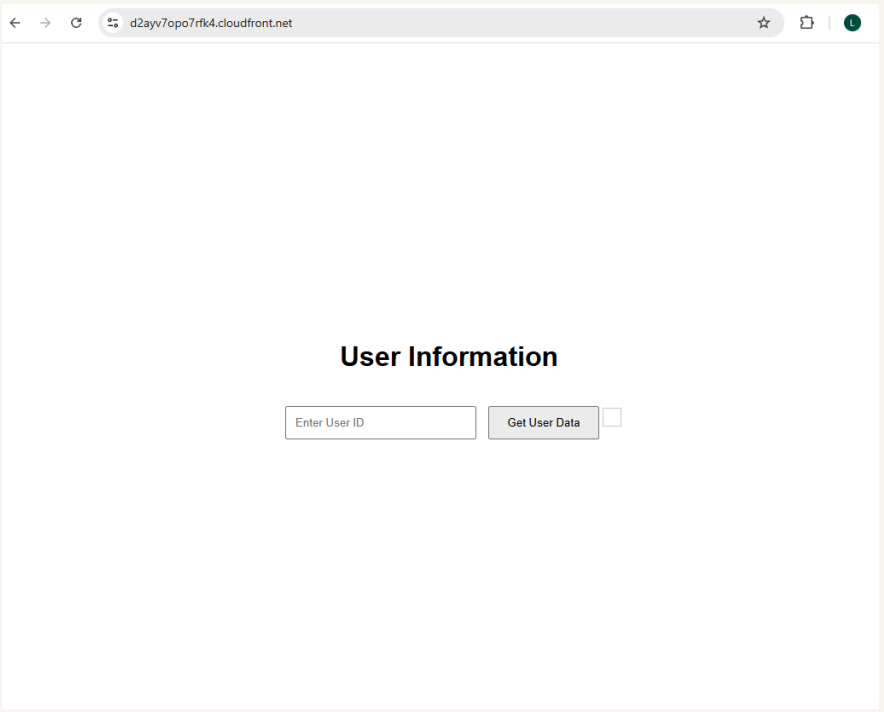
nextwork.org

I did this project today to gain hands-on experience building a real-world serverless application with AWS services. Yes, it met my goals because I now understand how frontend (CloudFront + S3) integrates with backend (API Gateway + Lambda + DynamoDB), and I learnt how to debug and fix CORS issues which are a common pain point in web development.

Presentation tier

For the presentation tier, I will set up an S3 bucket to store my website files and use CloudFront to deliver them globally because this ensures my website is easily accessible, fast, and scalable for users around the world.

I accessed my delivered website by using the CloudFront distribution domain name that was generated after connecting my S3 bucket to CloudFront. Instead of trying to open the S3 bucket URL directly (which would give an AccessDenied error because the bucket is private), I used the CloudFront URL (for example, https://d2ayv7opo7rfk4.cloudfront.net). This domain securely fetches the files from my S3 bucket through the permissions I configured in the bucket policy. After invalidating the cache to clear old errors, I was able to view my index.html file and confirm the website was loading correctly via CloudFront's global content delivery network.

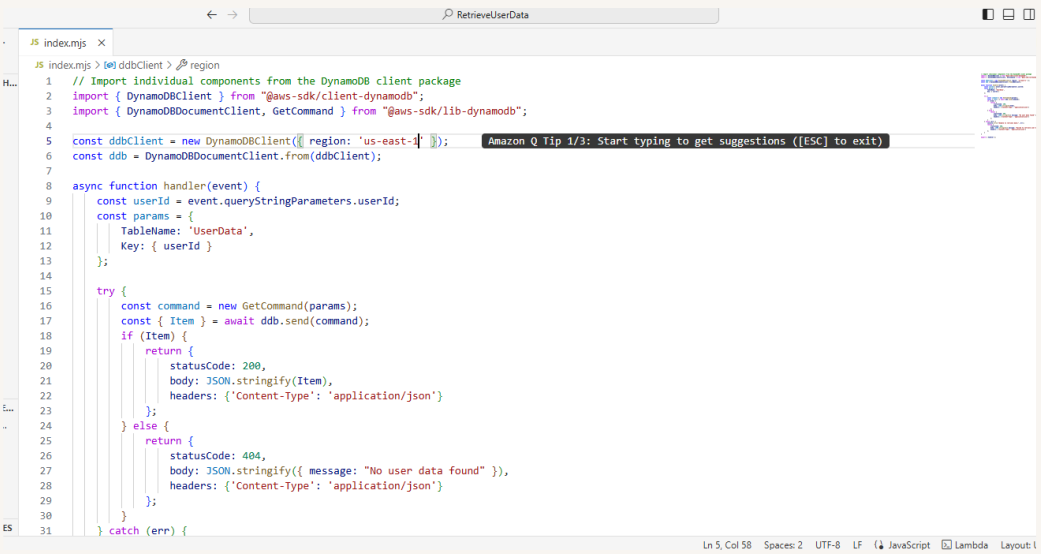

d2ayv7opo7rfk4.cloudfront.net
User Information
Enter User ID
Get User Data

Logic tier

For the logic tier, I will set up an AWS Lambda function connected to API Gateway because this tier handles the application's business logic. The Lambda function will fetch data from a DynamoDB table and return it to the requester. API Gateway will expose the Lambda function as a RESTful API endpoint, allowing external clients (like the presentation tier website) to make GET requests. This setup ensures that the logic tier acts as the "brains" of the application, processing requests and delivering dynamic data securely and efficiently.

The Lambda function retrieves data by accepting an event with a userId query parameter, then using the AWS SDK for JavaScript to query the UserData DynamoDB table.

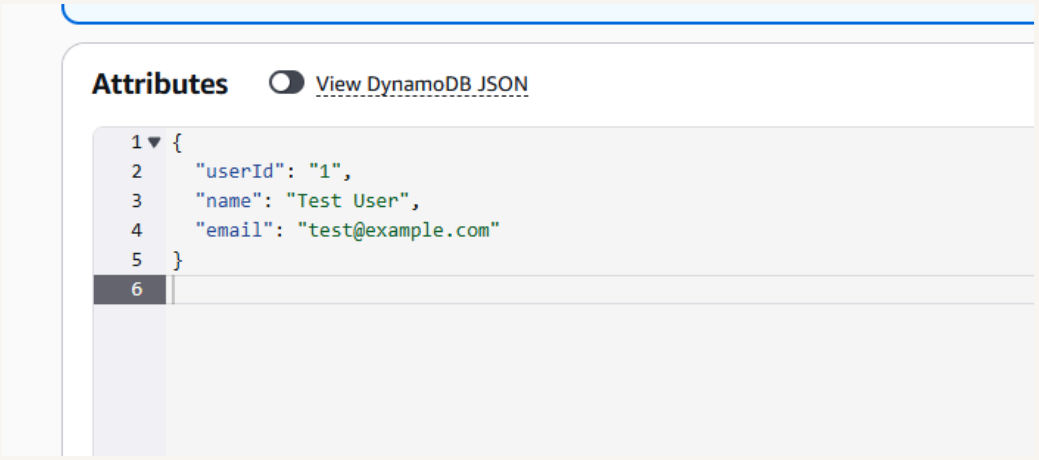
Louis Moyo
NextWork Student



Data tier

For the data tier, I will set up a DynamoDB table because it provides a fully managed, highly available NoSQL database that can scale automatically and is ideal for storing user data for my application. This table will hold user records that the Lambda function can query and return through the API.

The partition key for my DynamoDB table is userId, which means every record in the table will be uniquely identified by a userId value. This allows my Lambda function to quickly look up and retrieve the correct user data whenever a request is made.



Logic and Data tier

Once all three layers of my three-tier architecture are set up, the next step is to integrate them by updating my frontend code (script.js) so it can call the API Gateway endpoint. This is important because it connects the presentation tier (website) with the logic tier (Lambda) and the data tier (DynamoDB), allowing the application to display live user data instead of static content.

I tested my API by copying the Invoke URL from my API Gateway's prod stage, then appending /users?userId=1 to the end of it. I ran this full URL in my web browser, which returned the DynamoDB table's data. This confirmed that the logic tier (Lambda) and the data tier (DynamoDB) were integrated correctly.

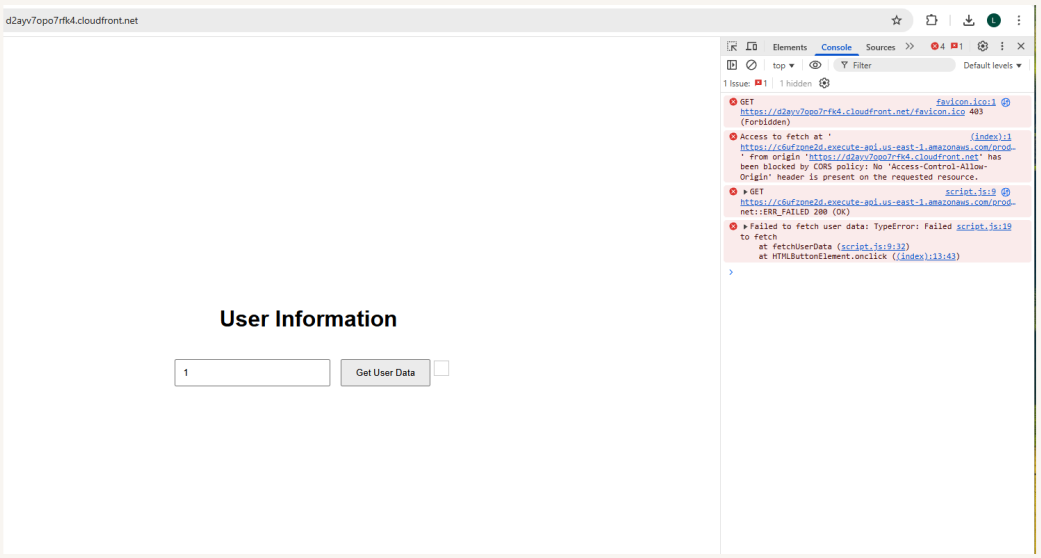

c6ufzpne2d.execute-api.us-east-1.amazonaws.com/prod/users?userId=1
Pretty print
{"email":"test@example.com","name":"Test User","userId":"1"}

Console Errors

The error in my distributed site was because the script.js file was still pointing to a placeholder API URL (https://[YOUR-PROD-API-URL]/users?userId=1) instead of my actual API Gateway invoke URL. Since the frontend was calling a non-existent endpoint, the request failed, causing the error in the browser console.

To resolve the error, I updated script.js by replacing the placeholder API URL with my actual API Gateway invoke URL. I then reuploaded it into S3 because the CloudFront-distributed website needs the updated JavaScript file to correctly call the API and display the user data.

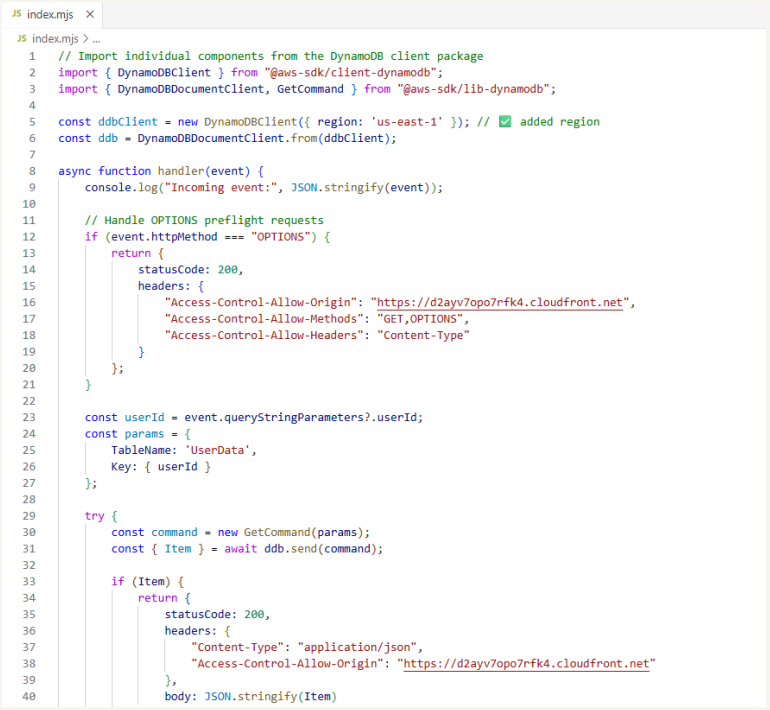
I ran into a second error after updating script.js. This was an error with CORS (Cross-Origin Resource Sharing) because my API Gateway wasn't configured to allow requests from my CloudFront distribution URL. By default, API Gateway only allows calls directly to its own Invoke URL, so when my CloudFront-hosted site tried to talk to it, the browser blocked the request. This happened because CORS headers weren't enabled on the API Gateway method/response.


d2ayv7opo7rfk4.cloudfront.net
User Information
1
Get User Data
Elements
Console
Sources
Filter
Default levels
1 Issue:
1 hidden
GET
favicon.ico:1
https://d2ayv7opo7rfk4.cloudfront.net/favicon.ico 403
(Forbidden)
Access to fetch at '
(index):1
https://c6ufzone2d.execute-api.us-east-1.amazonaws.com/prod-
' from origin 'https://d2ayv7opo7rfk4.cloudfront.net' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested resource.
GET
script.js:9
https://c6ufzone2d.execute-api.us-east-1.amazonaws.com/prod-
net::ERR_FAILED 200 (OK)
Failed to fetch user data: TypeError: Failed
script.js:19
to fetch
at fetchUserData (script.js:9:32)
at HTMLButtonElement.onclick ((index):13:43)

Resolving CORS Errors

To resolve the CORS error, I first went back into API Gateway and enabled CORS on the /users resource. I configured both GET and OPTIONS methods under Access-Control-Allow-Methods, since the browser sends an OPTIONS preflight request before a real GET call. I also set my CloudFront distribution domain name as the Access-Control-Allow-Origin value. This updated my API Gateway to explicitly allow requests coming from my CloudFront-hosted site.

I also updated my Lambda function because the CORS headers need to be set at the response level, not just in API Gateway. The changes I made were adding the AWS region (us-east-1) explicitly for DynamoDB, handling the OPTIONS preflight request with proper headers, and restricting Access-Control-Allow-Origin to my CloudFront domain instead of using *. This ensures that my CloudFront-hosted frontend can securely call my API without running into CORS errors.



index.mjs

```
// Import individual components from the DynamoDB client package
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const ddbClient = new DynamoDBClient({ region: 'us-east-1' }); // ✅ added region
const ddb = DynamoDBDocumentClient.from(ddbClient);

async function handler(event) {
    console.log("Incoming event:", JSON.stringify(event));

    // Handle OPTIONS preflight requests
    if (event.httpMethod === "OPTIONS") {
        return {
            statusCode: 200,
            headers: {
                "Access-Control-Allow-Origin": "https://d2ayv7opo7rfk4.cloudfront.net",
                "Access-Control-Allow-Methods": "GET,OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"
            }
        };
    }

    const userId = event.queryStringParameters?.userId;
    const params = {
        TableName: 'UserData',
        Key: { userId }
    };

    try {
        const command = new GetCommand(params);
        const { Item } = await ddb.send(command);

        if (Item) {
            return {
                statusCode: 200,
                headers: {
                    "Content-Type": "application/json",
                    "Access-Control-Allow-Origin": "https://d2ayv7opo7rfk4.cloudfront.net"
                },
                body: JSON.stringify(Item)
```

Fixed Solution

I verified the fixed connection between API Gateway and CloudFront by redeploying my Lambda changes, clearing the CloudFront cache, and then testing the site again through the CloudFront domain. Using the browser's developer tools, I confirmed that the network requests returned status code 200 and that the response headers included the correct Access-Control-Allow-Origin set to my CloudFront URL, which proved that the CORS issue was resolved.

The place to learn & showcase your skills

Check out nextwork.org for more projects