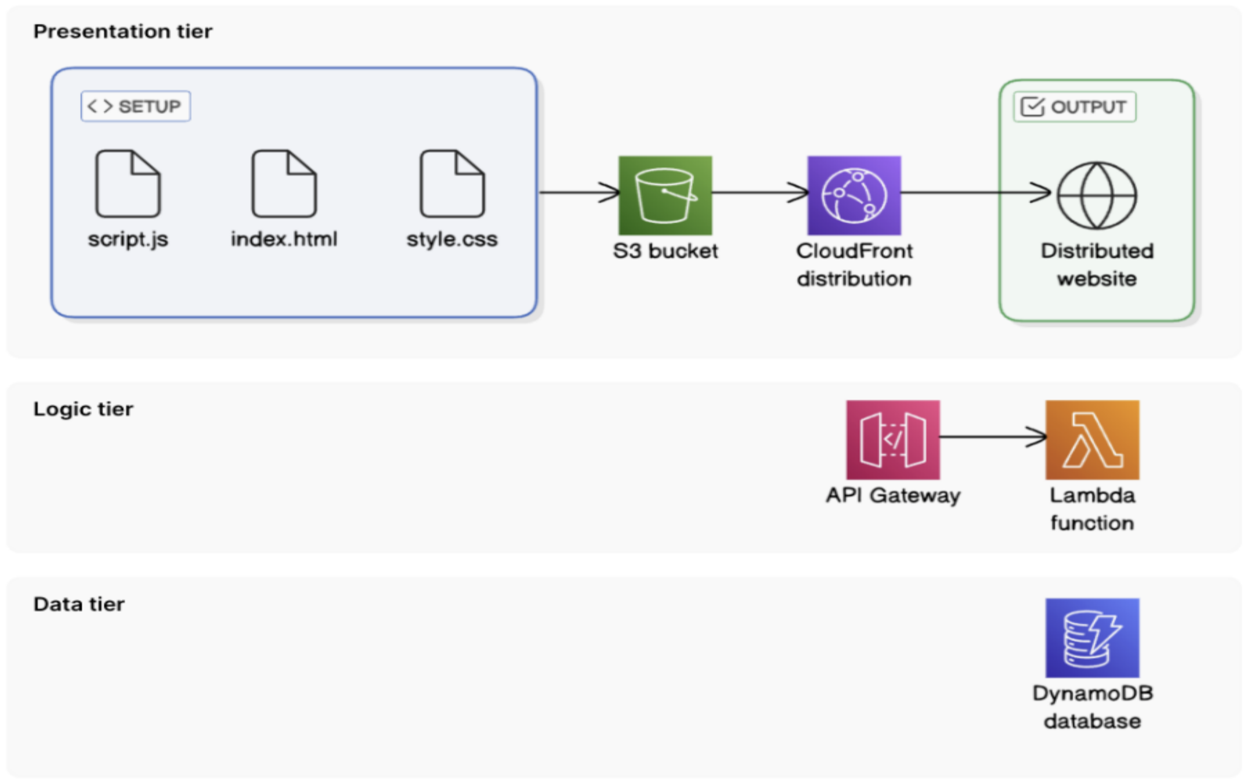
# AWS Three-Tier Web Application

## Introduction

In this project, I will demonstrate how to setup a three-tier web app from scratch! I'm doing this project to learn how to connect all the three tier. I will start with presentation tier then setup logic tier, last the data tier before tying them all.

I did this project today to learn about the three-tier architecture and setup my web app. This project absolutely met my goals and I could see a fully functioning web app by the end.

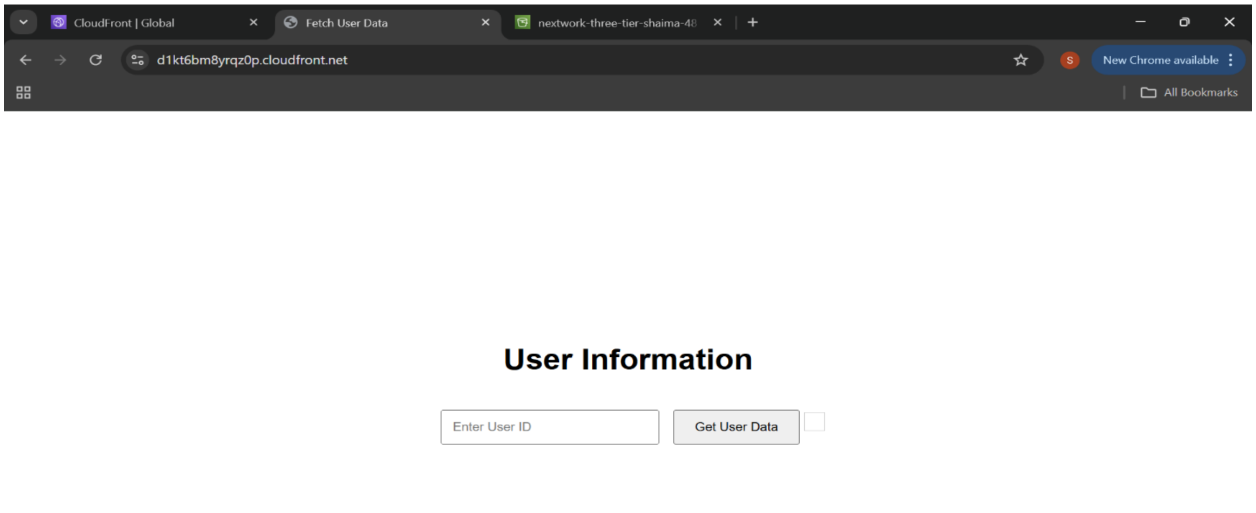


## Tools and concepts

Services I used Amazon S3, CloudFront, DynamoDB, Lambda, API Gateway. Key concepts I learnt include Lambda functions, CORS errors, updating the Javascript file with the API Invoke URL, and testing the invoked URL within the browser.

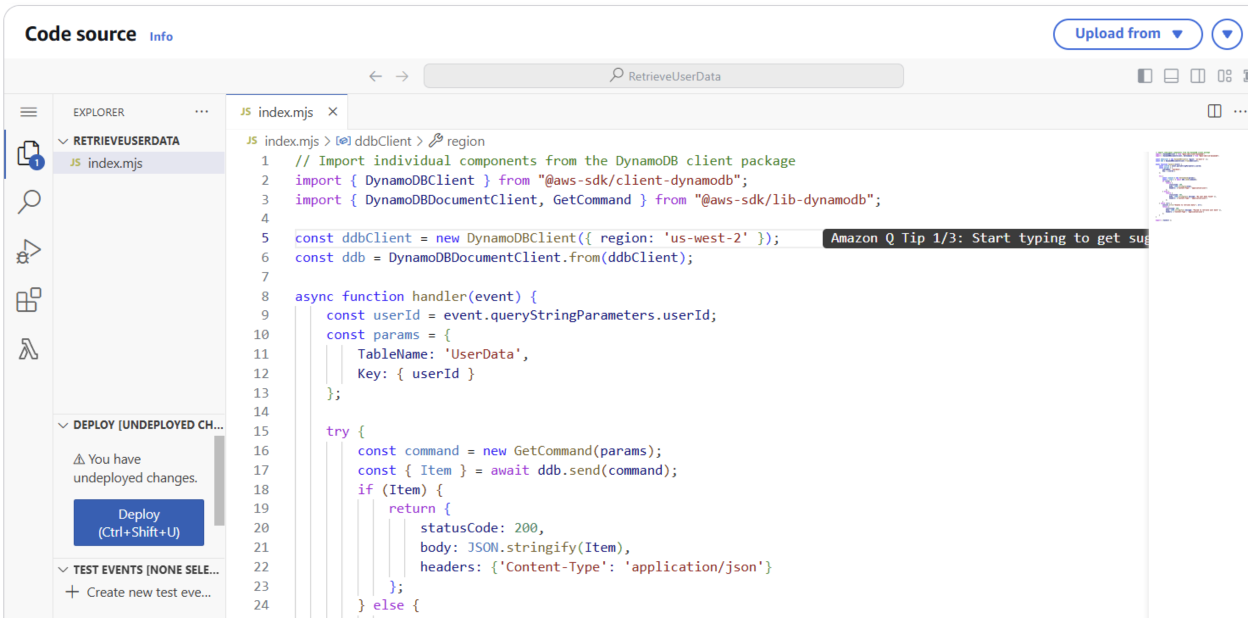
## Presentation tier

For the presentation tier, I will setup how my website will be displayed and available to end users. This is because the presentation tier is responsible for storing my website's files(Amazon S3) + website distribution (Amazon Cloudfront). I accessed my delivered website by visiting the Cloudfront distribution's URL. This URL works because I also setup an origin access control that lets my S3 bucket restrict access to only my CloudFront distribution.



## Logic tier

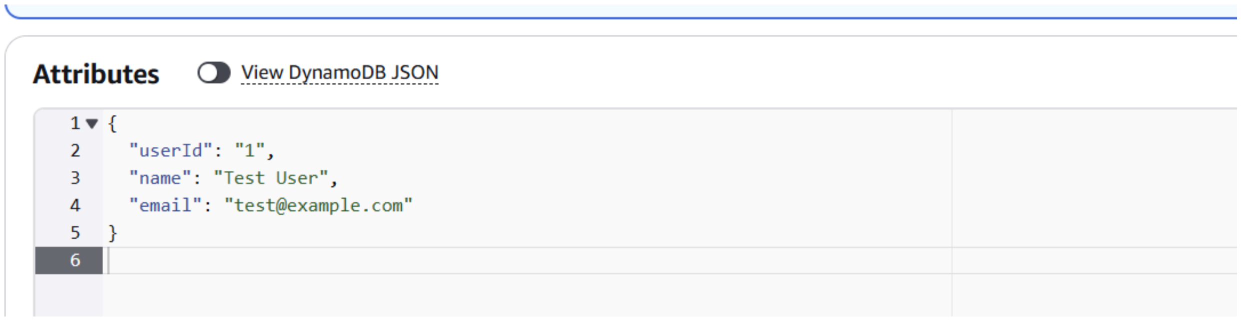
For the logic tier, I will set up a Lambda function to handle requests (e.g. look up userId to return some user data) and also an API with API Gateway to receive requests from the user and hand it over to Lambda. The Lambda function retrieves data by looking up a userId (that the user enters over the web app) in DynamoDB. The AWS SDK is used in the function code so we can use templates and libraries that lets us find the correct DynamoDB table + request data.



```
// 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-west-2' });
const ddb = DynamoDBDocumentClient.from(ddbClient);

async function handler(event) {
    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,
                body: JSON.stringify(Item),
                headers: {'Content-Type': 'application/json'}
            };
        } else {
```

## Data tier

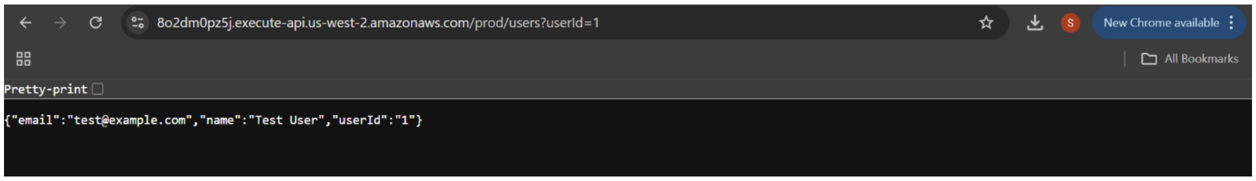
For the data tier, I will set up a DynamoDB database that stores user data. At the moment there is no user data for us to return to our web app's users. The data in my database will get returned once I set up DynamoDB and connect it will Lambda The partition key for my DynamoDB table is userId. This means that when my table looks up user data, it will look it up based om userId. Then, it can return all data(values) related to the item with that ID.



```
{
  "userId": "1",
  "name": "Test User",
  "email": "test@example.com"
}
```

## Logic and Data tier

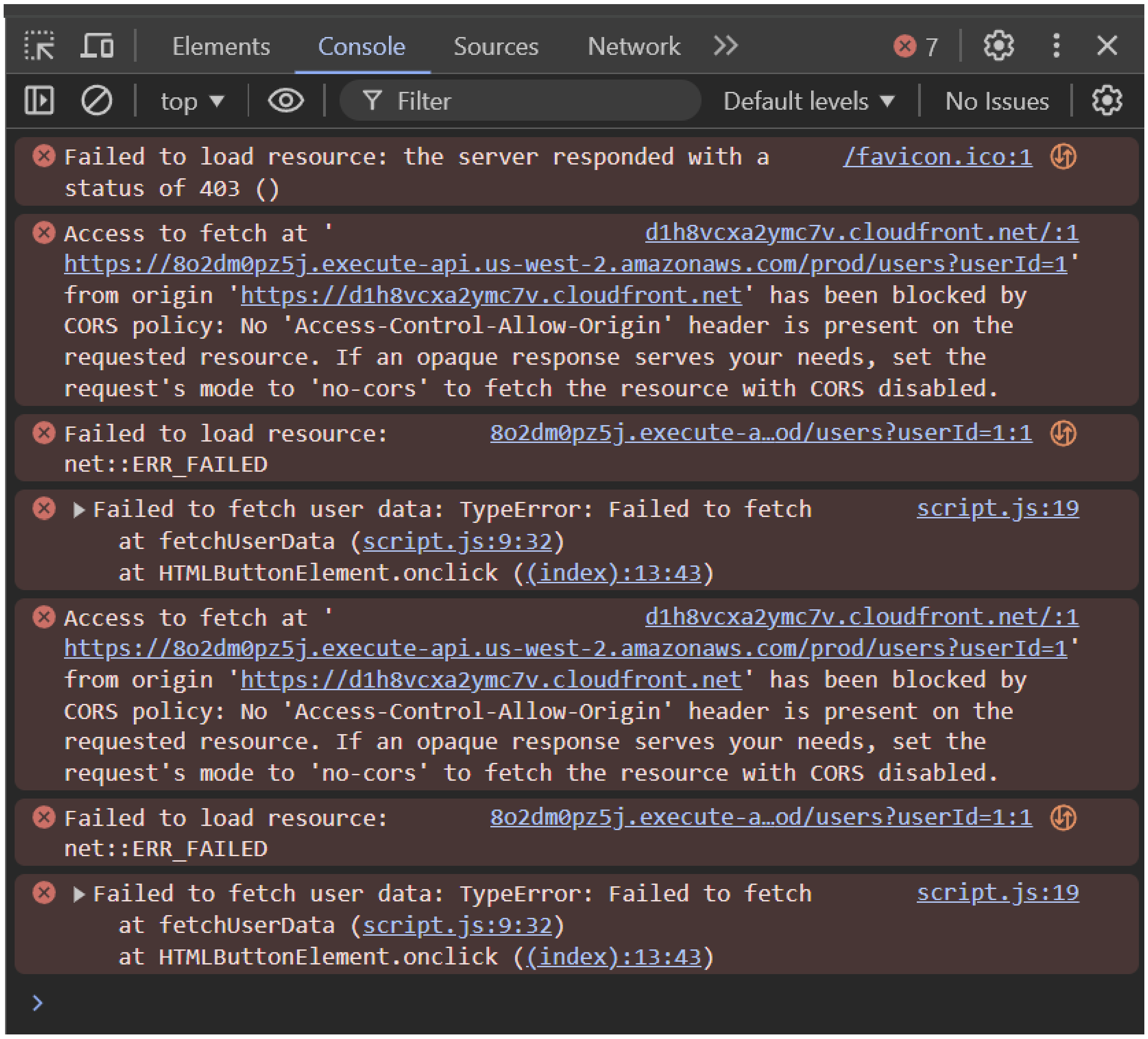
Once all three layers of my three-tier architecture are set up, the next step is to connect the presentation and logic tier. This is because currently there is no way for my API to catch requests that users make through our distributed site! To test my API, I visited the Invoke URL of the prod stage API. This lets me test whether I can use the API and retrieve user data. The results were some user data in JSON when we looked up userId=1. This proved a logic+ data tier connection.


8o2dm0pz5j.execute-api.us-west-2.amazonaws.com/prod/users?userId=1
New Chrome available
All Bookmarks
Pretty-print

```
{"email":"test@example.com","name":"Test User","userId":"1"}
```

## Console Errors

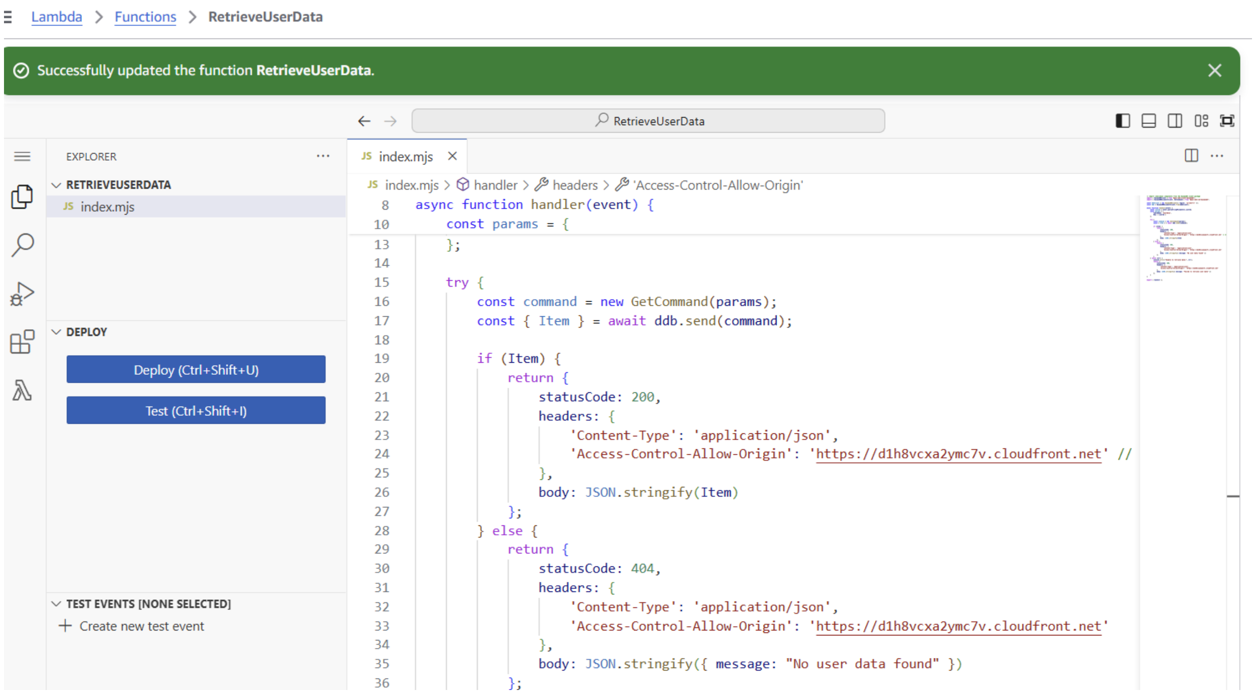
The error in my distributed site was because there was an error within script.js (one of the website files I had uploaded into S3). The script.js file is referencing an prod stage API URL placeholder and not my API's actual URL. To resolve the error, I updated script.js by replacing some placeholder text with the API's prod stage URL. I then reuploaded it into S3 because the S3 bucket is still storing the last uploaded version (i.e with the error). I ran into a second error after updating script.js. This was an error on the browser side with CORS because API Gateway, by default, is only configured to allow requests from users directly running its Invoke URL in the browser (not with CloudFront).



```
Failed to load resource: the server responded with a status of 403 ()    /favicon.ico:1

Access to fetch at 'https://8o2dm0pz5j.execute-api.us-west-2.amazonaws.com/prod/users?userId=1' from origin 'https://d1h8vcxa2ymc7v.cloudfront.net' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.    d1h8vcxa2ymc7v.cloudfront.net/:1

Failed to load resource: net::ERR_FAILED    8o2dm0pz5j.execute-a…od/users?userId=1:1

▶ Failed to fetch user data: TypeError: Failed to fetch    script.js:19
    at fetchUserData (script.js:9:32)
    at HTMLButtonElement.onclick ((index):13:43)

Access to fetch at 'https://8o2dm0pz5j.execute-api.us-west-2.amazonaws.com/prod/users?userId=1' from origin 'https://d1h8vcxa2ymc7v.cloudfront.net' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.    d1h8vcxa2ymc7v.cloudfront.net/:1

Failed to load resource: net::ERR_FAILED    8o2dm0pz5j.execute-a…od/users?userId=1:1

▶ Failed to fetch user data: TypeError: Failed to fetch    script.js:19
    at fetchUserData (script.js:9:32)
    at HTMLButtonElement.onclick ((index):13:43)
```

## Resolving CORS Errors

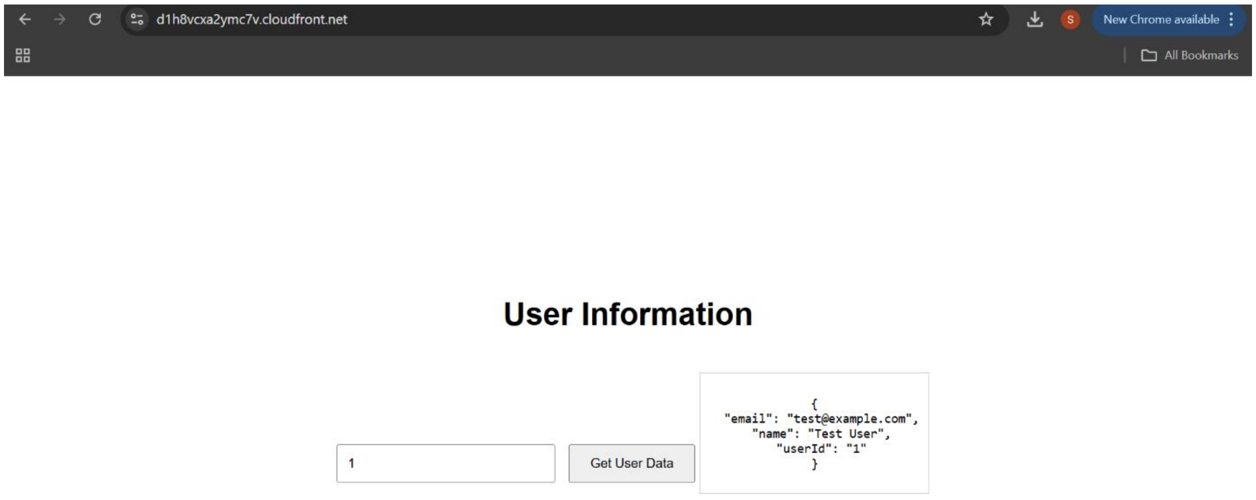
To resolve the CORS error, I first went into my API(in API Gateway) and enabled CORS on the /user resource. I then made sure GET requests are enabled, and referenced my CloudFront domain as the domain getting access. I also updated my Lambda function because it needs to be able to return CORS headers to show that it has the permission to invoke API's URL and return a response. I added 'Access-Control-Allow-Origin' as a header in the response.


Lambda > Functions > RetrieveUserData
Successfully updated the function RetrieveUserData.
RetrieveUserData
EXPLORER
RETRIEVEUSERDATA
index.mjs
DEPLOY
Deploy (Ctrl+Shift+U)
Test (Ctrl+Shift+I)
TEST EVENTS [NONE SELECTED]
Create new test event
index.mjs > handler > headers > 'Access-Control-Allow-Origin'

```
async function handler(event) {
    const params = {
    };

    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://d1h8vcxa2ymc7v.cloudfront.net' //
                },
                body: JSON.stringify(Item)
            };
        } else {
            return {
                statusCode: 404,
                headers: {
                    'Content-Type': 'application/json',
                    'Access-Control-Allow-Origin': 'https://d1h8vcxa2ymc7v.cloudfront.net'
                },
                body: JSON.stringify({ message: "No user data found" })
            };
```

## Fixed Solution

I verified the fixed connection between API Gateway and CloudFront by looking up user data in the distributed site again. In our final test, user data could be returned- so a user request in the presentation tier gets data from the data tier.



```
{
"email": "test@example.com",
"name": "Test User",
"userId": "1"
}
```