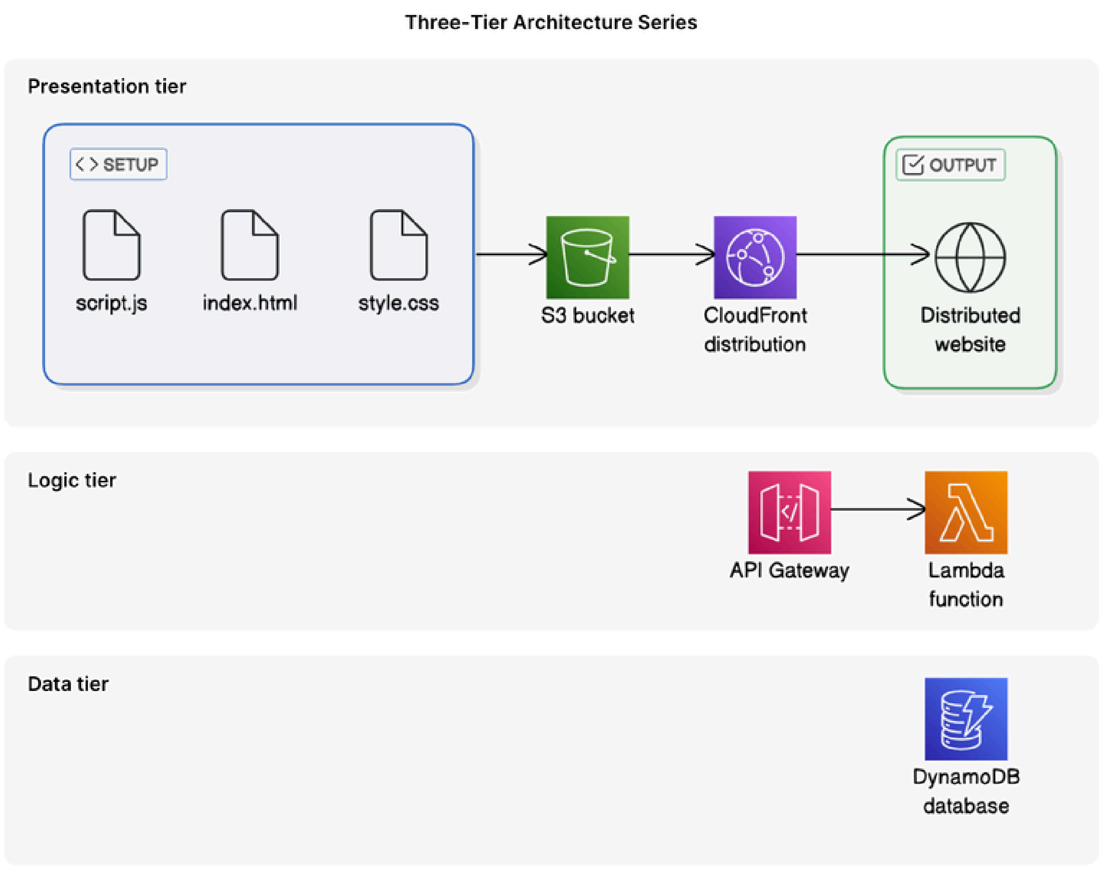
Kareshma Rajaananthapadmanaban

Build Three-Tier Web App

Build and deploy a complete three-tier web application on AWS by integrating S3, CloudFront, Lambda, API Gateway, and DynamoDB covering everything from static hosting to dynamic data retrieval.

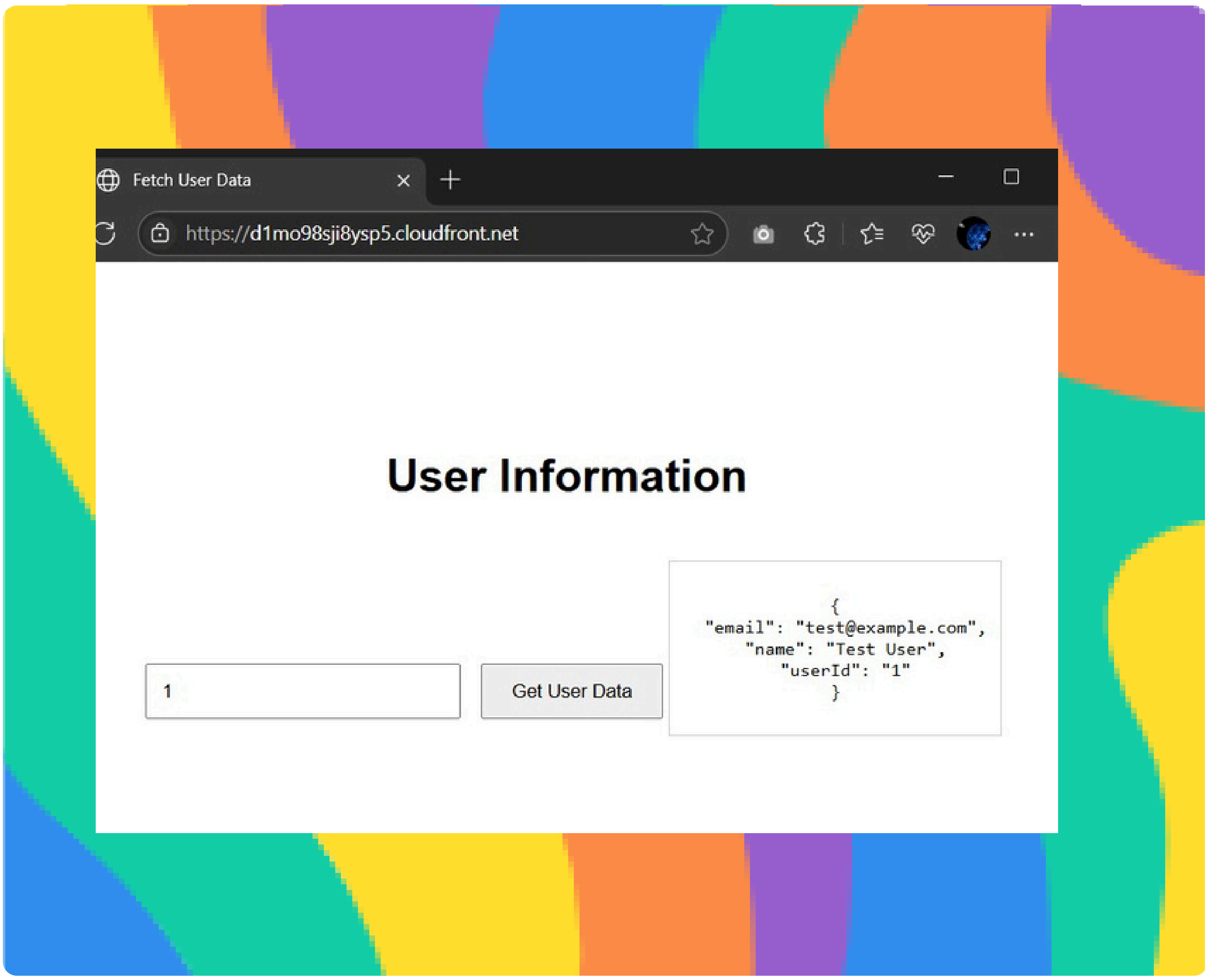


nextwork.org

Build a Three-Tier Web App

Kareshma Rajaananthapadmanaban

Learn to connect, test, and secure each layer of a scalable, serverless architecture.



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

Introducing Today's Project!

In this project, I will demonstrate how to build a **fully functional three-tier web application** using AWS services. I'm doing this project to learn how to integrate the presentation, logic, and data tiers using S3, CloudFront, Lambda, API Gateway, and DynamoDB in a scalable, serverless architecture.

Tools and concepts

Services I used were **S3**, **CloudFront**, **Lambda**, **API Gateway**, and **DynamoDB**. Key concepts I learnt include Lambda functions for serverless compute, **REST APIs** with API Gateway, storing and retrieving data using DynamoDB, **CORS** handling, and how to connect the presentation, logic, and data tiers to build a fully functional web app on AWS.

Project reflection

This project took me approximately 1 hour to complete, plus a few extra minutes to write the documentation.The most challenging part was troubleshooting the **API connection** and **resolving CORS-related errors**. It was most rewarding to see the entire three-tier architecture working seamlessly and displaying live data on the website.

I did this project today to practice building a **full-stack serverless web application** using real AWS services. My goal was to understand how the presentation, logic, and data tiers work together in a cloud-native architecture. Yes, this project met my goals it helped me apply core AWS concepts hands-on and gave me confidence in deploying a complete three-tier solution.

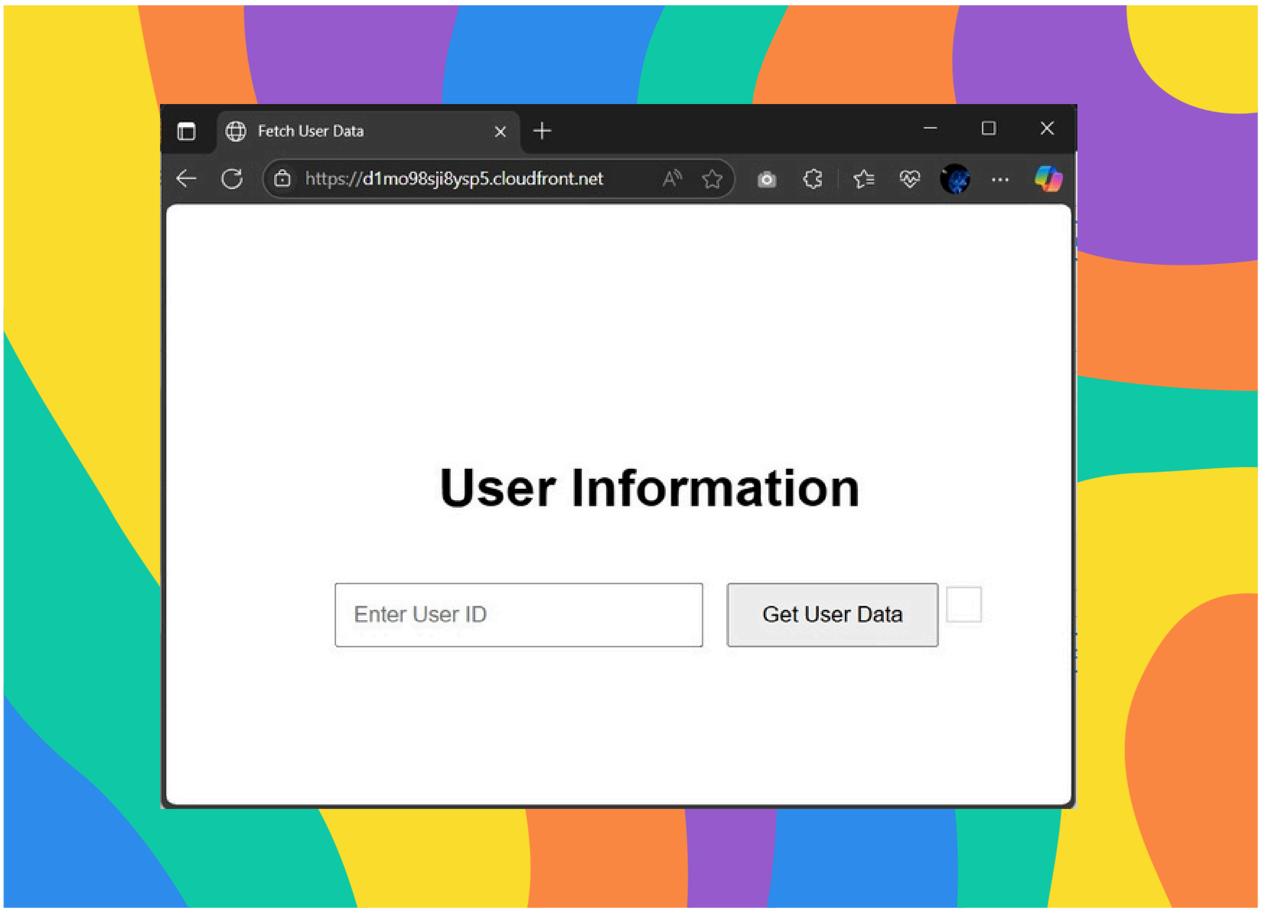
Presentation tier

To begin building the presentation tier, I will create the frontend layer of the application using **static website files**. Hosting these files in an S3 bucket gives me durable and cost-effective storage, while pairing it with CloudFront helps deliver content with low latency and high availability to users worldwide.

For the presentation tier, I will set up an **S3 bucket** to host my static website files and use **CloudFront** to distribute the content globally, because this ensures fast, reliable, and secure delivery of the web interface to users across different regions.

After uploading my website assets (HTML, CSS, JS) to the S3 bucket, I set the **bucket policy** to allow public read access and enabled static website hosting. This made the contents of the bucket accessible via a web browser, allowing CloudFront to retrieve and serve those files to users.

I accessed my delivered website by copying the domain name from my CloudFront distribution and pasting it into my browser. CloudFront fetches the files stored in my S3 bucket and delivers them through its **global edge locations**. This ensures that my website **loads faster** for users around the world by serving cached content from the nearest server, making the experience smooth and reliable.



Website loaded through CloudFront.

To improve performance and security, I enabled **caching** and **HTTPS** in CloudFront. This ensures that static assets are served quickly and securely, even during high traffic. Updating the site is simple just re-upload files to the S3 bucket, and CloudFront handles the global delivery with **minimal delay**.

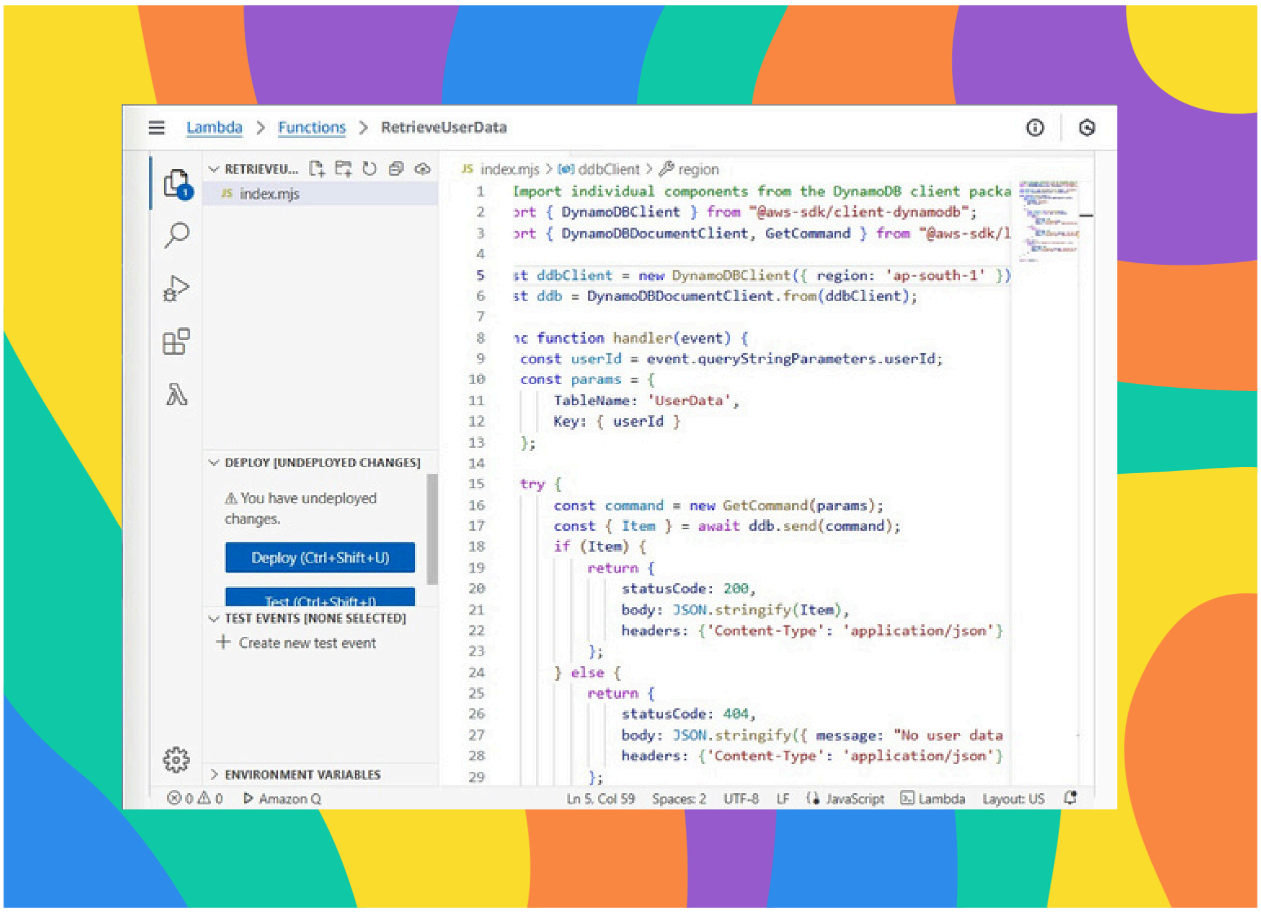
Logic tier

To build the logic tier, I will use AWS Lambda to **run backend code** in response to HTTP requests managed by API Gateway. This eliminates the need to manage servers and enables scalable execution of application logic whenever a user triggers an action from the frontend.

For the logic tier, I will set up an AWS Lambda function connected to an API Gateway because this layer handles the **core application logic.** It processes incoming user requests, fetches data from the DynamoDB table, and returns responses. This setup enables scalable, serverless interaction between the frontend and backend.

I configured the API Gateway to **trigger** the Lambda function when a **GET** request is made to a specific endpoint. The API acts as a secure entry point, validating and routing incoming HTTP requests to the backend logic. This setup allows controlled access to serverless functions from the web interface.

The Lambda function retrieves data by using the **AWS SDK** to connect to DynamoDB. It extracts the `userId` from the incoming HTTP requests' query string parameters. Then, it uses **`GetCommand`** to look up the item in the `UserData` table with that `userId`. If the item is found, it returns it in the response with a 200 status code. If not, it returns a 404. Errors during the process return a 500 status with an appropriate error message.


Lambda > Functions > RetrieveUserData
RETRIEVEU...
index.mjs
DEPLOY [UNDEPLOYED CHANGES]
⚠ You have undeployed changes.
Deploy (Ctrl+Shift+U)
Test (Ctrl+Shift+I)
TEST EVENTS [NONE SELECTED]
+ Create new test event
ENVIRONMENT VARIABLES
index.mjs > ddbClient > region
⊗ 0 ⚠ 0 Amazon Q
Ln 5, Col 59 Spaces: 2 UTF-8 LF JavaScript Lambda Layout: US

```
Import individual components from the DynamoDB client packa
ort { DynamoDBClient } from "@aws-sdk/client-dynamodb";
ort { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/l

st ddbClient = new DynamoDBClient({ region: 'ap-south-1' })
st ddb = DynamoDBDocumentClient.from(ddbClient);

nc 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 {
         return {
             statusCode: 404,
             body: JSON.stringify({ message: "No user data
             headers: {'Content-Type': 'application/json'}
         };
```

Lambda function code

Using Lambda and API Gateway allows me to decouple the business logic from the frontend. This makes the system modular and easier to maintain or extend later. It also helps with cost-efficiency, as I **only pay** when the **function** is **triggered**, making it ideal for event-driven workloads.

Data tier

In a three-tier web architecture, the data tier is responsible for **securely storing** and **retrieving application data**. Since this layer powers all dynamic interactions, it's critical to choose a storage service that is fast, scalable, and integrates well with serverless components. DynamoDB fits these needs perfectly in our serverless setup.

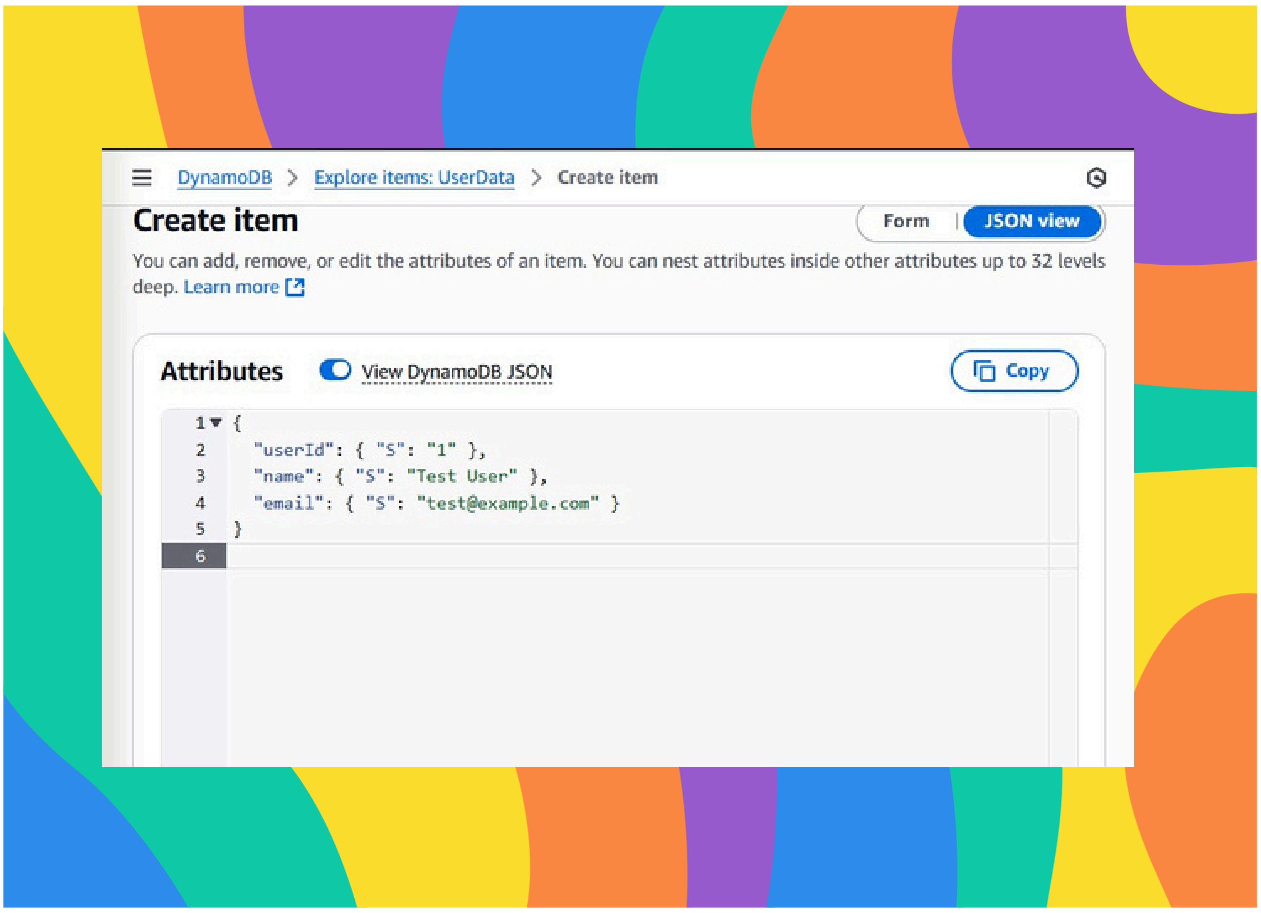
For the data tier, I will set up a DynamoDB table to store and manage user data because it's a fast, **serverless NoSQL database** that integrates well with Lambda. This allows my web app to retrieve user-specific information dynamically through the API, making the application functional and data-driven.

The partiton key for my DynamoDB table is `userId`, which means each item in the table is uniquely identified by a user ID. This lets me efficiently retrieve individual user data. I'm using DynamoDB to store **structured user records** that my Lambda function can fetch when requested through the API.

Once the table status changes to Active, I selected the UserData table and chose Explore table items. I clicked Create item, switched to JSON view, turned off View DynamoDB JSON, and added the following data:

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

This sample item will help **test data retrieval** through Lambda.



Create item

You can add, remove, or edit the attributes of an item. You can nest attributes inside other attributes up to 32 levels deep. Learn more

Attributes · View DynamoDB JSON

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

Data item adding to table

After adding the sample item, I granted my Lambda function **read access** to DynamoDB. In the Permissions tab of the Lambda console, I selected the **execution role** and attached the AmazonDynamoDBReadOnlyAccess policy. Now the Lambda function can securely read data from the UserData table our data tier is ready and connected!

To ensure **reliable backend operations**, I verified that the Lambda function could successfully query the DynamoDB table and return accurate data through the API Gateway. This confirmed the seamless link between data storage and application logic, completing a robust serverless backend foundation for my web application.

Logic and Data tier

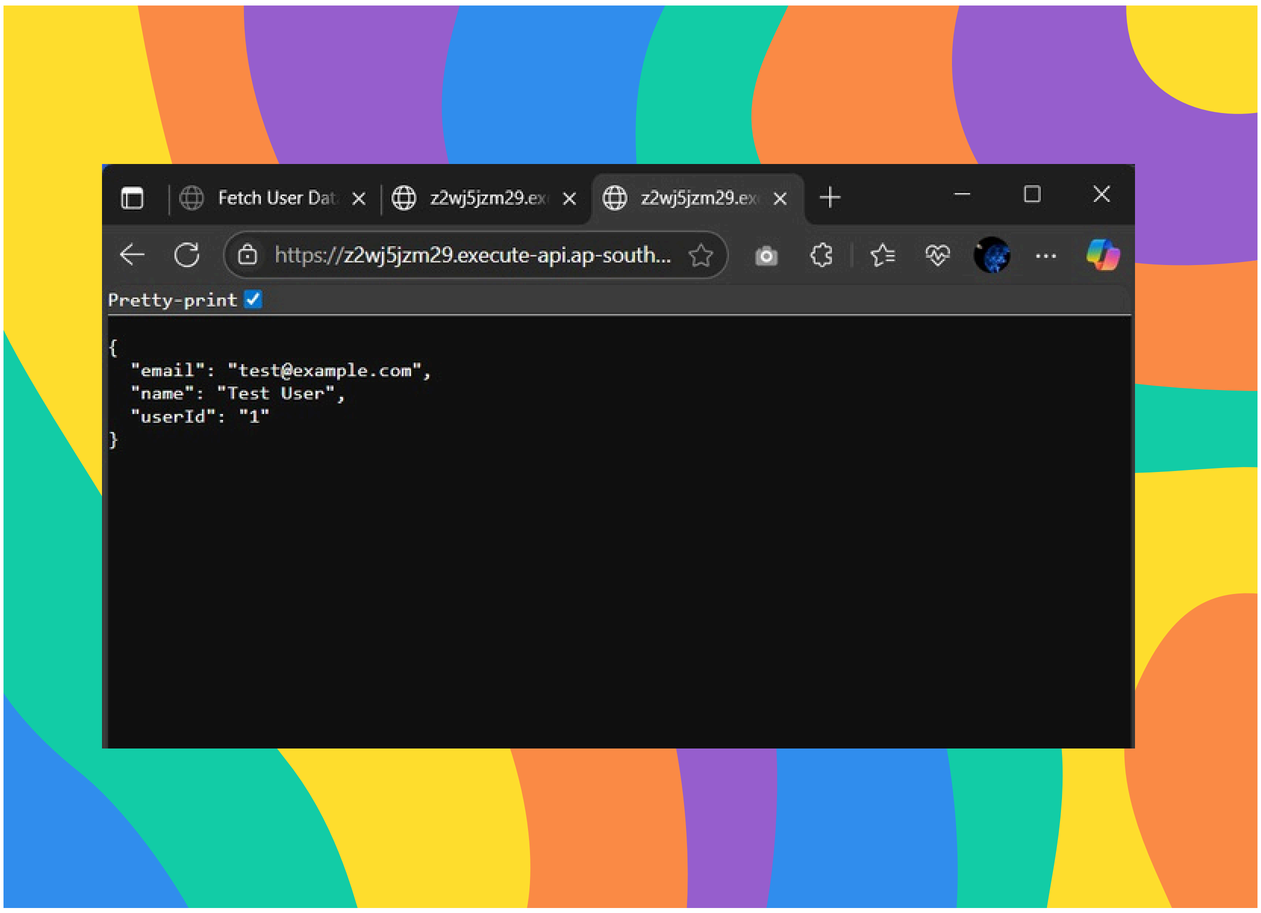
Once all **three layers** of my three-tier architecture are set up, the next step is to connect them by updating the frontend (presentation tier) to talk to the backend API (logic tier). This ensures that when a user visits the site, the browser can **fetch** and **display** data from DynamoDB through API Gateway and Lambda, completing the app flow.

Logic and data tier's integration verified

To test my API, I copied the Invoke URL from the API Gateway's prod stage and **appended** `` `/users?userId=1` `` to it. I then opened the full URL in my browser. The results were successful I saw the expected user data returned from the DynamoDB table, which confirmed that the API Gateway and Lambda function integration is working correctly.

why I can't see any user data yet

It's likely happening because the `` `script.js` `` file hasn't been updated yet to actually call the API Gateway endpoint or the updated file hasn't been uploaded to the S3 bucket again after editing. Another possibility is a **CORS issue**: if the API Gateway wasn't configured to allow requests from the CloudFront domain, the browser would block the response.

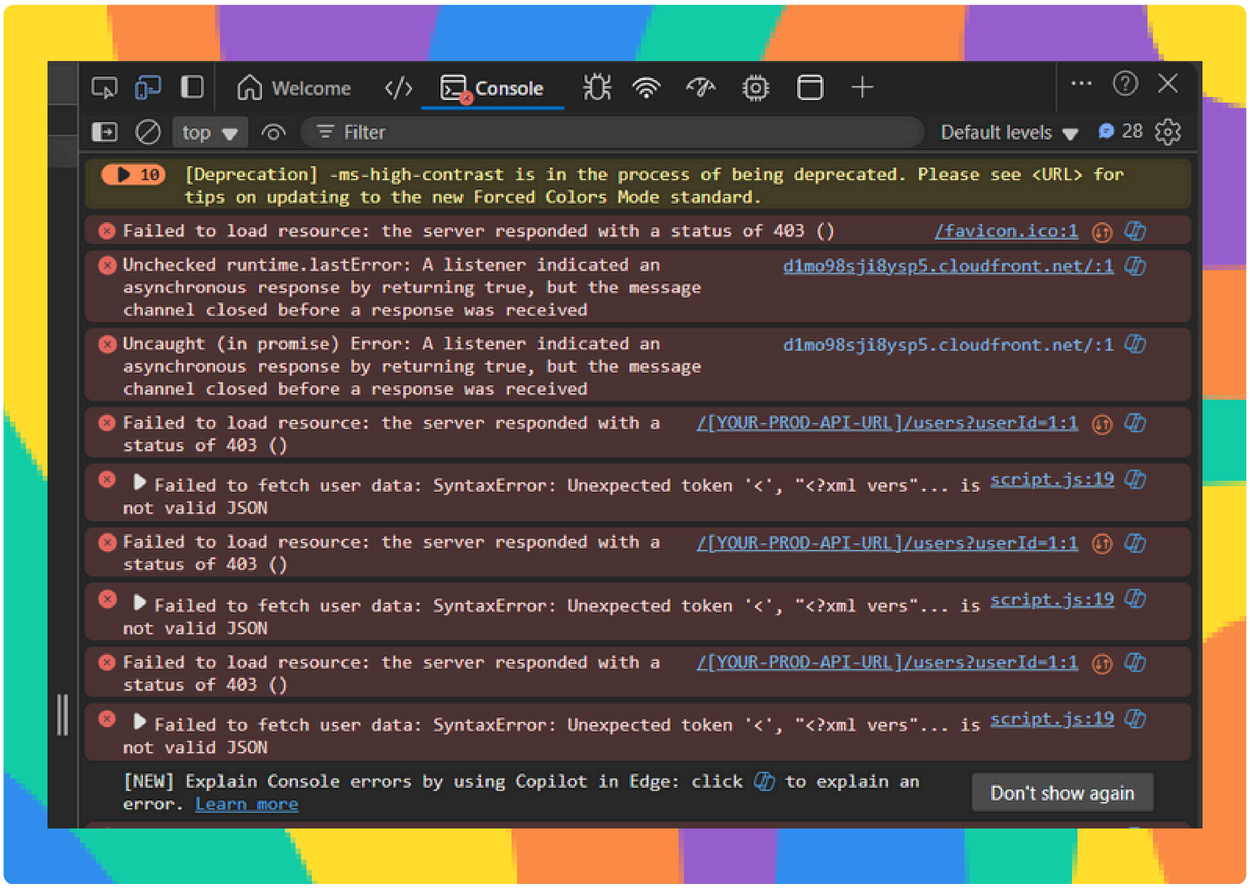


```
{
  "email": "test@example.com",
  "name": "Test User",
  "userId": "1"
}
```

API resuts after integrating Logic and Data tier

Console Errors

The error in my distributed site was because the `**script.js**` file still had a placeholder `[YOUR-PROD-API-URL]` instead of my actual API Gateway invoke URL. As a result, the site was trying to make a request to a non-existent endpoint, which caused a JavaScript error and prevented any data from being fetched or displayed.



```
10 [Deprecation] -ms-high-contrast is in the process of being deprecated. Please see <URL> for
   tips on updating to the new Forced Colors Mode standard.
Failed to load resource: the server responded with a status of 403 ()          /favicon.ico:1
Unchecked runtime.lastError: A listener indicated an           d1mo98sji8ysp5.cloudfront.net/:1
asynchronous response by returning true, but the message
channel closed before a response was received
Uncaught (in promise) Error: A listener indicated an           d1mo98sji8ysp5.cloudfront.net/:1
asynchronous response by returning true, but the message
channel closed before a response was received
Failed to load resource: the server responded with a     /[YOUR-PROD-API-URL]/users?userId=1:1
status of 403 ()
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is script.js:19
not valid JSON
Failed to load resource: the server responded with a     /[YOUR-PROD-API-URL]/users?userId=1:1
status of 403 ()
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is script.js:19
not valid JSON
Failed to load resource: the server responded with a     /[YOUR-PROD-API-URL]/users?userId=1:1
status of 403 ()
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is script.js:19
not valid JSON
[NEW] Explain Console errors by using Copilot in Edge: click to explain an
error. Learn more
```

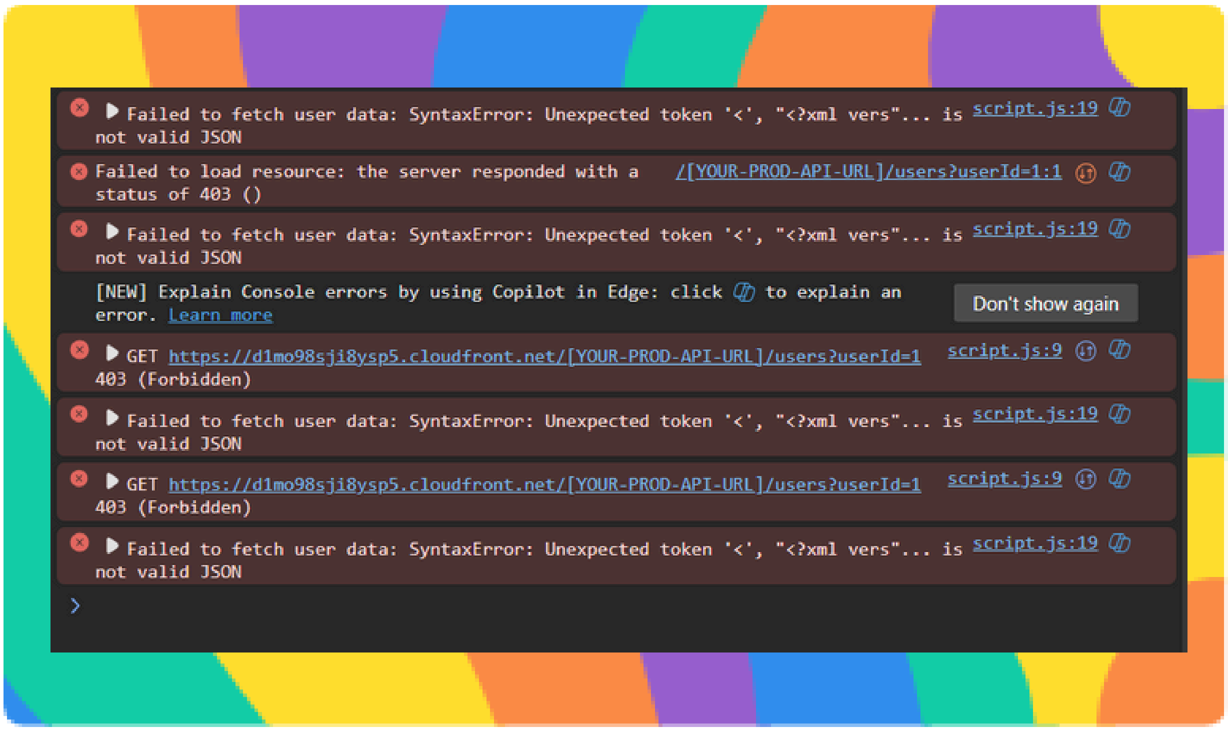
Failed to load resource

To resolve the error, I **updated** script.js by replacing the `[YOUR-PROD-API-URL]` placeholder with my actual API Gateway invoke URL. I then reuploaded it into S3 because CloudFront serves files from the S3 bucket, so the updated JavaScript needed to be present there for the website to correctly fetch and display user data.

Resolving CORS Errors

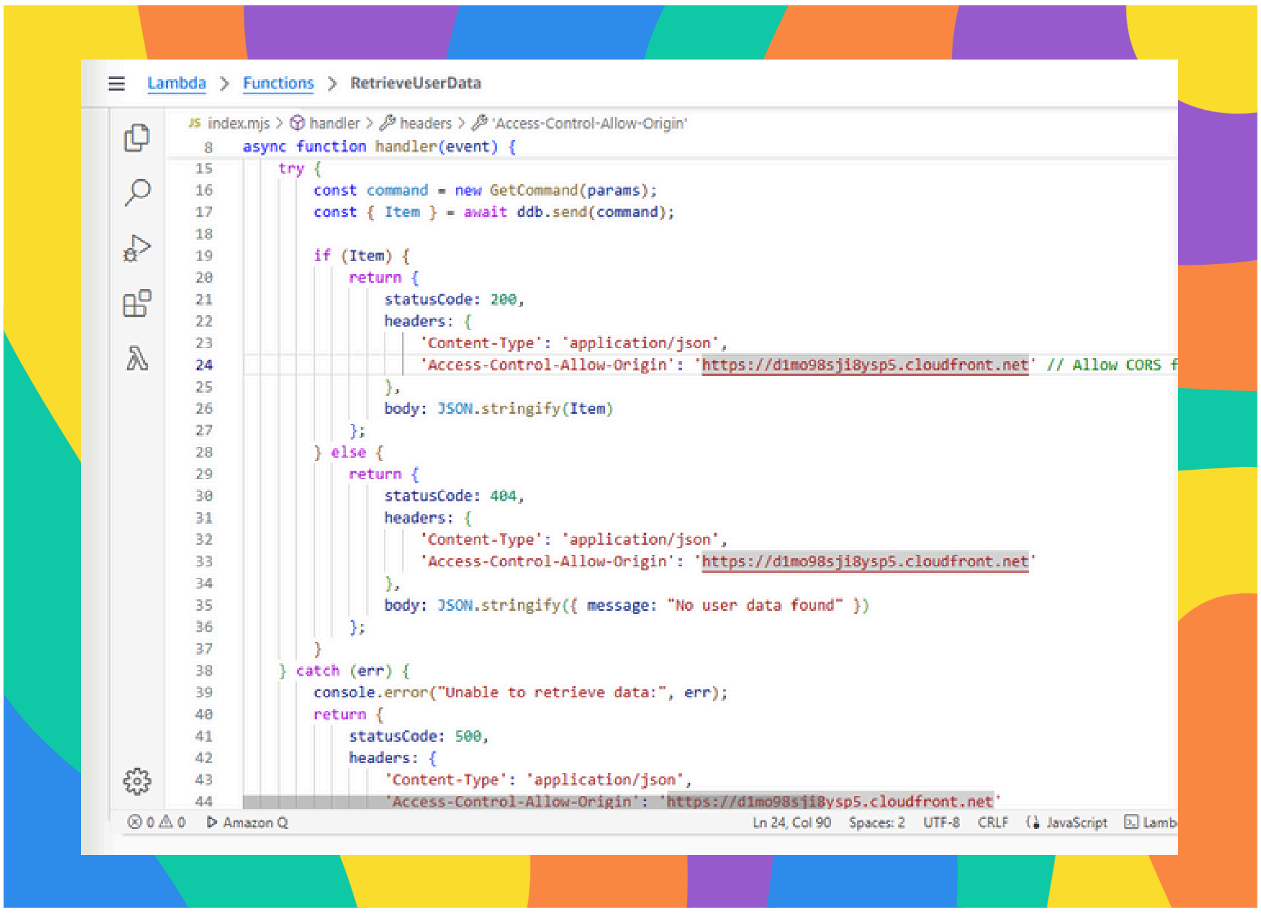
In this step, I will validate that my web app works end-to-end by testing the CloudFront-distributed website again, because I want to ensure that the presentation, logic, and data tiers are fully integrated and functioning correctly. This final validation confirms that user input can successfully trigger API requests and return database results.

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 **accept requests** from the CloudFront-distributed frontend. Without proper CORS settings, the browser blocks cross-origin requests between the frontend and backend.



```
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is   script.js:19
not valid JSON
Failed to load resource: the server responded with a   /[YOUR-PROD-API-URL]/users?userId=1:1
status of 403 ()
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is   script.js:19
not valid JSON
[NEW] Explain Console errors by using Copilot in Edge: click  to explain an
error. Learn more
▶ GET https://d1mo98sji8ysp5.cloudfront.net/[YOUR-PROD-API-URL]/users?userId=1   script.js:9
403 (Forbidden)
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is   script.js:19
not valid JSON
▶ GET https://d1mo98sji8ysp5.cloudfront.net/[YOUR-PROD-API-URL]/users?userId=1   script.js:9
403 (Forbidden)
▶ Failed to fetch user data: SyntaxError: Unexpected token '<', "<?xml vers"... is   script.js:19
not valid JSON
>
```

403 Forbidden error while verifying CloudFront site


Lambda > Functions > RetrieveUserData
index.mjs > handler > headers > 'Access-Control-Allow-Origin'

```
async function handler(event) {
  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://d1mo98sji8ysp5.cloudfront.net' // Allow CORS f
        },
        body: JSON.stringify(Item)
      };
    } else {
      return {
        statusCode: 404,
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': 'https://d1mo98sji8ysp5.cloudfront.net'
        },
        body: JSON.stringify({ message: "No user data found" })
      };
    }
  } catch (err) {
    console.error("Unable to retrieve data:", err);
    return {
      statusCode: 500,
      headers: {
        'Content-Type': 'application/json',
        'Access-Control-Allow-Origin': 'https://d1mo98sji8ysp5.cloudfront.net'
```

⊗ 0 ⚠ 0   Amazon Q   Ln 24, Col 90   Spaces: 2   UTF-8   CRLF   JavaScript   Lamb

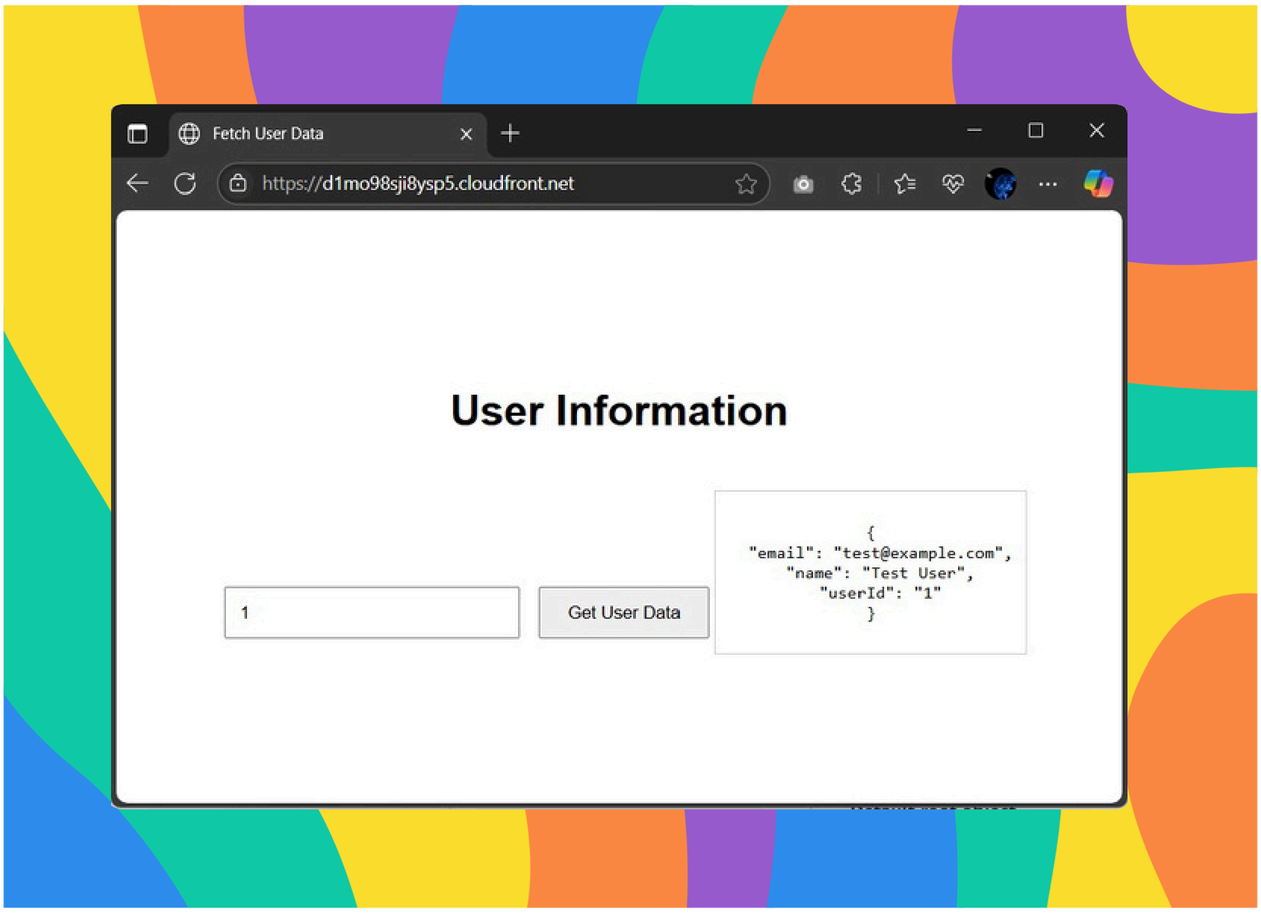
Updated Lambda function

To resolve the CORS error, I first **enabled CORS** on my API Gateway's `/users` resource. I selected both GET and OPTIONS methods under Access-Control-Allow- Methods and added my CloudFront distribution domain under Access-Control-Allow- Origin. This allows my frontend to securely interact with the backend API.

I also updated my Lambda function because I'm using Lambda Proxy Integration, which requires the function itself to return CORS headers. The changes I made were adding `'Access-Control-Allow-Origin': **'*'**` in the response headers so that the frontend (hosted on CloudFront) can successfully make API requests without being blocked by the browser's CORS policy.

Fixed Solution

I verified the fixed connection between API Gateway and CloudFront by refreshing my CloudFront URL and testing the user data retrieval. This time, the data was successfully fetched from DynamoDB and displayed on the website, confirming that the **CORS configuration** and **Lambda response headers** were correctly updated.



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

Website displaying the data from DynamoDB.

Delete the Resources

To avoid ongoing charges, delete all resources created in this project:

1. **CloudFront Distribution**
    - Go to CloudFront console → select distribution → Disable
    - Wait until status is Disabled → Delete
2. **S3 Bucket**
    - Go to S3 console → select your bucket
    - Empty it first → then Delete
3. **API Gateway**
    - Go to API Gateway → select your API
    - From Actions, choose Delete API
4. **Lambda Function**
    - Go to Lambda console → select your function
    - Under Actions, choose Delete
5. **DynamoDB Table**
    - Go to DynamoDB → Tables → select UserData
    - Select Delete, type confirm
    - ✅ Leave Delete CloudWatch alarms checked
    - ❌ Leave On-demand backup unchecked

⚠️ Tip: Always delete CloudFront first before S3 to avoid errors.

Kareshma Rajaananthapadmanaban

Follow up for more interesting projects !!!

Check out nextwork.org / my profile for more projects