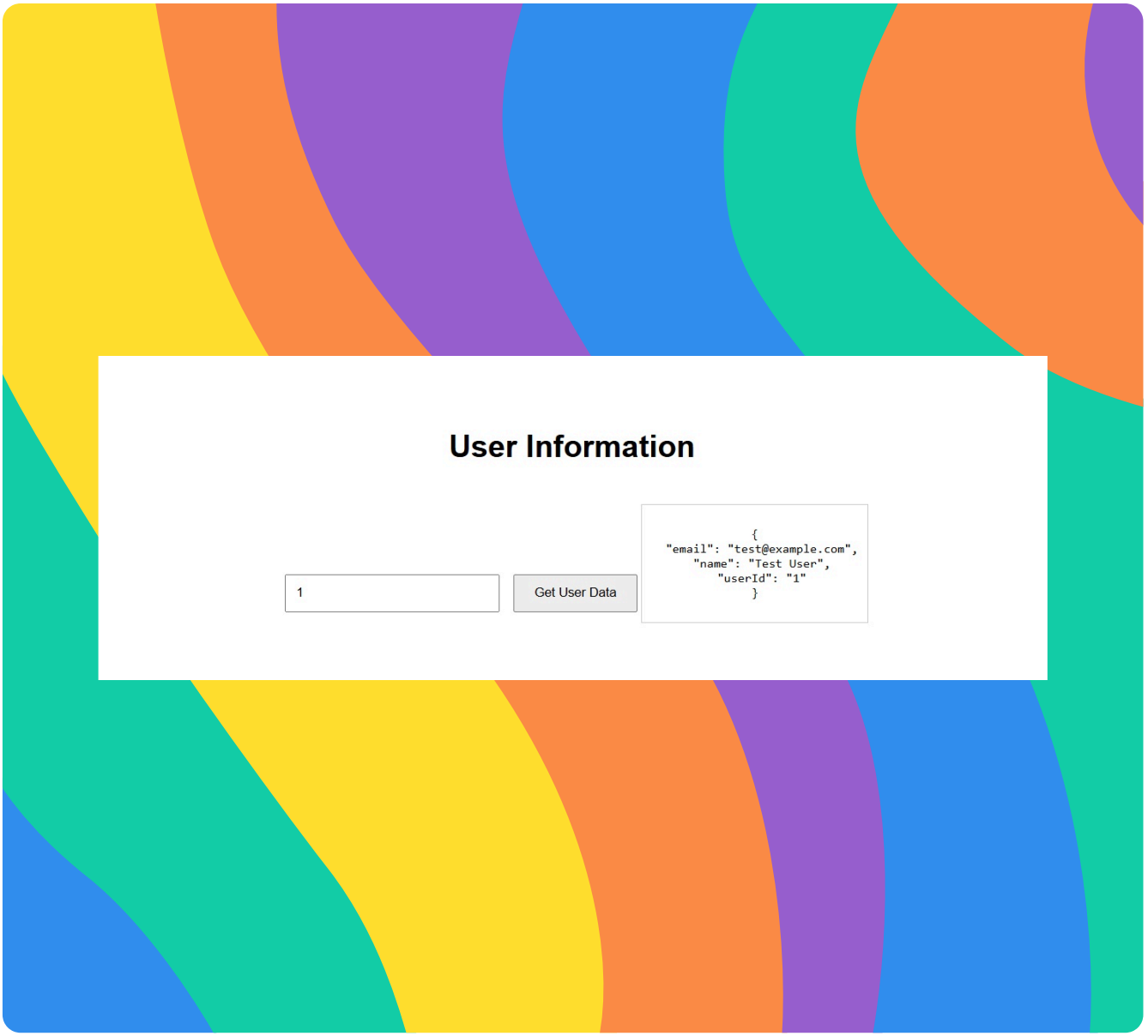
nextwork.org

Build a Three-Tier Web App

DI Dineshraj Dhanapathy



DI
Dineshraj Dhanapathy
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to build a scalable, serverless three-tier web application using AWS services. I'm doing this project to learn how to store website files in S3, distribute content globally with CloudFront, handle backend logic with Lambda, manage APIs with API Gateway, and store user data using DynamoDB. By connecting all these services, I'll gain hands-on experience in building modern cloud-native applications.

Tools and concepts

Services I used were Amazon S3, CloudFront, Lambda, API Gateway, and DynamoDB. Key concepts I learnt include Lambda functions for serverless logic, API Gateway for creating and managing RESTful APIs, CORS configuration for secure cross-origin requests, and integrating frontend and backend layers in a three-tier architecture. I also gained hands-on experience in connecting AWS services, managing permissions, and debugging real-world errors like CORS issues and data fetch failures.

DI
Dineshraj Dhanapathy
NextWork Student

Project reflection

This project took me approximately 3 hours to complete. The most challenging part was troubleshooting the CORS errors and ensuring all services were properly connected with the correct permissions and configurations. It was most rewarding to see the user data successfully fetched from DynamoDB and displayed on my CloudFront-hosted website, confirming that my three-tier architecture was fully functional and integrated.

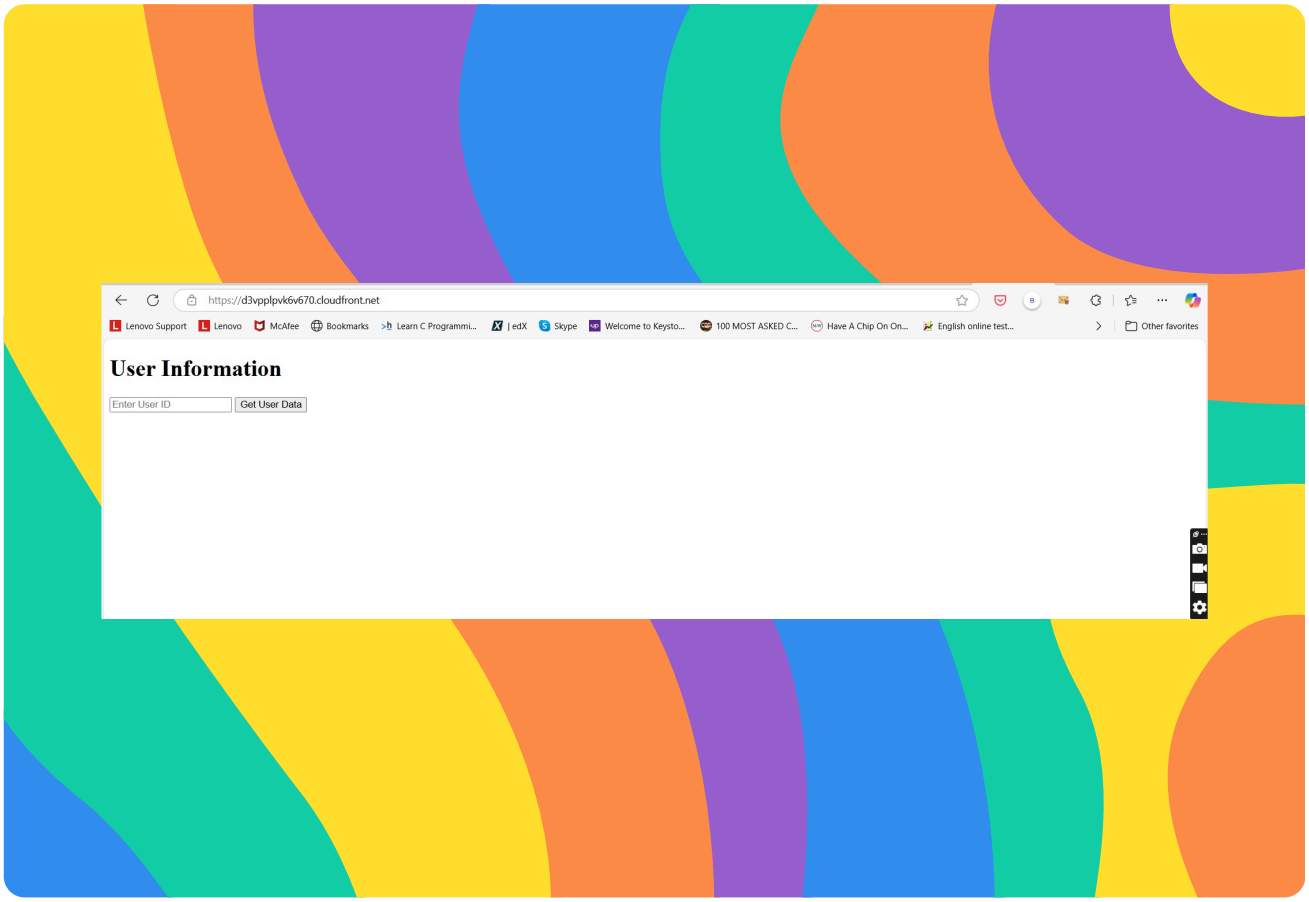
I did this project today to deepen my understanding of building a full-stack serverless application using AWS services. I wanted hands-on experience with integrating the three-tier architecture—presentation, logic, and data tiers—using tools like S3, CloudFront, Lambda, API Gateway, and DynamoDB. This project definitely met my goals by helping me learn how these services work together, how to troubleshoot issues like CORS, and how to build scalable, cloud-native applications efficiently.

DI
Dineshraj Dhanapathy
NextWork Student

Presentation tier

For the presentation tier, I will set up an S3 bucket to store my website's files and configure CloudFront to deliver content globally because this layer is responsible for displaying the site to users quickly and reliably. Hosting the index.html in S3 ensures a scalable and cost-effective solution, while CloudFront speeds up content delivery by caching it at edge locations worldwide, improving performance and user experience.

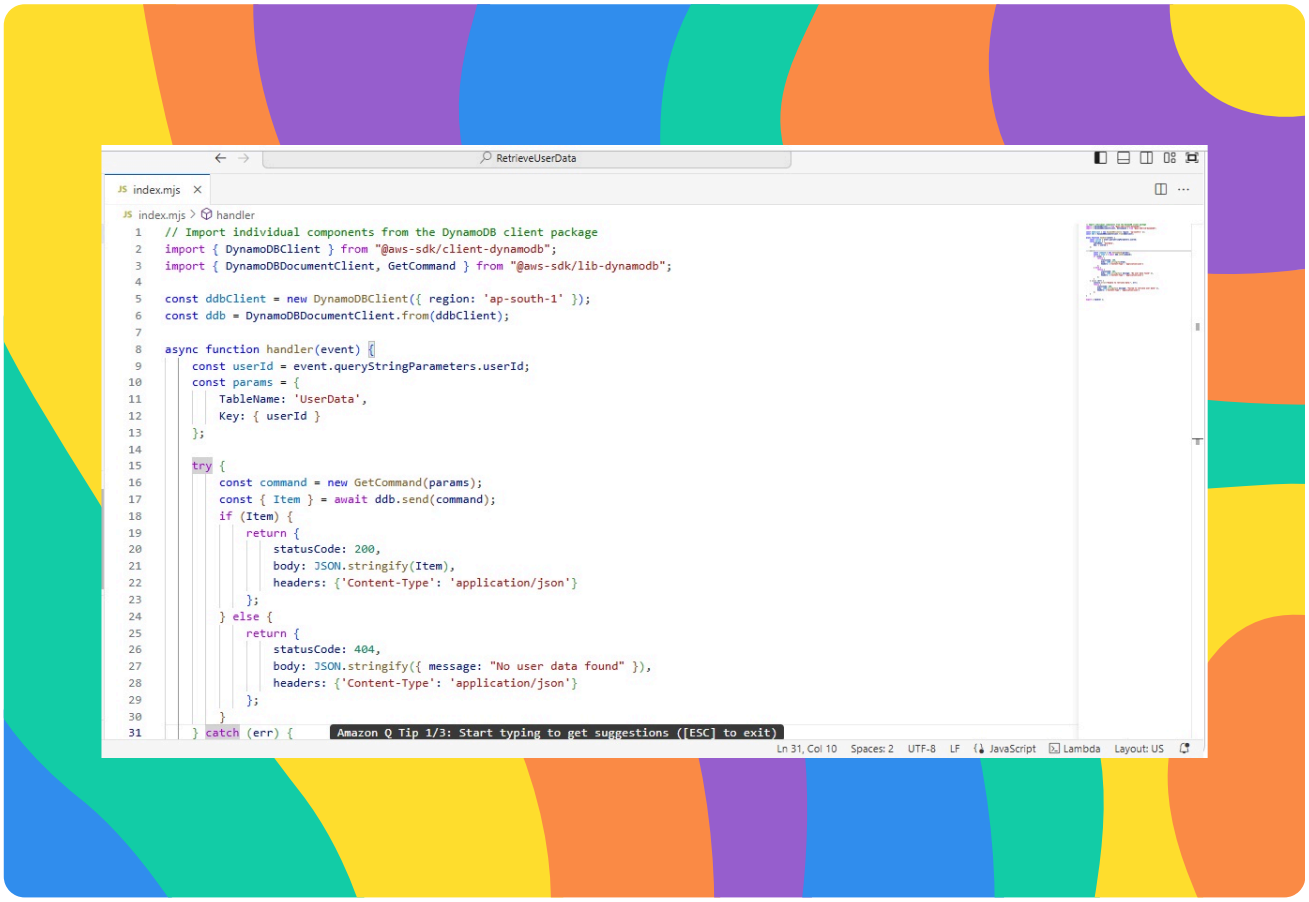
I accessed my delivered website by using the CloudFront distribution URL provided after setup. This URL points to the cached version of my index.html file stored in the S3 bucket. CloudFront ensures that the website loads quickly by delivering content from the nearest edge location, giving users a fast and responsive experience when interacting with the site's interface.


https://d3vpplpvk6v670.cloudfront.net
Lenovo Support
Lenovo
McAfee
Bookmarks
Learn C Programmi...
edX
Skype
Welcome to Keysto...
100 MOST ASKED C...
Have A Chip On On...
English online test...
Other favorites
User Information
Enter User ID
Get User Data

Logic tier

For the logic tier, I will set up a Lambda function to handle the backend logic of my application because it allows me to run code without managing servers. This function will fetch data from a DynamoDB table, acting as the brain of my app by processing requests and retrieving necessary information. I'll also create an API Gateway REST API to expose the Lambda function through a secure HTTP endpoint. By setting up a resource and method to handle GET requests and deploying the API, I make my logic tier accessible and efficient for real-time data interactions.

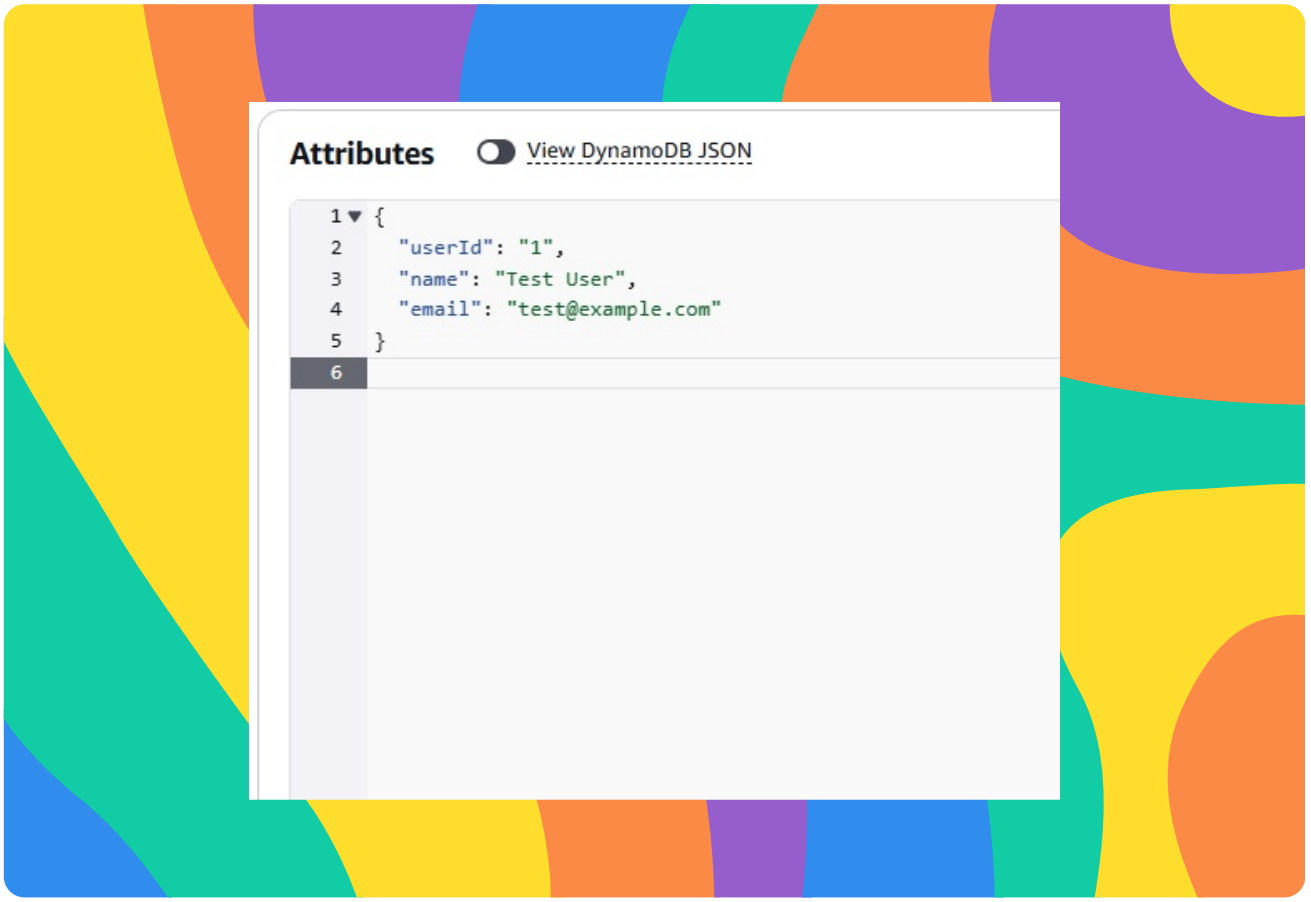
The Lambda function retrieves data by connecting to the DynamoDB table using the AWS SDK. When triggered by an API Gateway GET request, it runs code that queries or scans the table based on the parameters provided. The function uses permissions defined in its IAM role to securely access the DynamoDB resource, fetch the required data, and return it as a response to the API call. This serverless setup enables fast and scalable data retrieval without managing any infrastructure.


RetrieveUserData
index.mjs
index.mjs > handler
// 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: 'ap-south-1' });
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 {
return {
statusCode: 404,
body: JSON.stringify({ message: "No user data found" }),
headers: {'Content-Type': 'application/json'}
};
}
} catch (err) {
Amazon Q Tip 1/3: Start typing to get suggestions ([ESC] to exit)
Ln 31, Col 10
Spaces: 2
UTF-8
LF
JavaScript
Lambda
Layout: US

Data tier

For the data tier, I will set up a DynamoDB table to store and manage user data because it provides a fast, scalable, and serverless NoSQL database solution. By adding user data into the table, my application can retrieve and display information through the Lambda function, completing the backend workflow. This setup ensures that the app can handle large amounts of data efficiently with low latency and high availability.

The partition key for my DynamoDB table is userId, which means each item in the table is uniquely identified by a user ID. This allows efficient retrieval of user-specific data, as DynamoDB uses the partition key to quickly locate the item. We're using DynamoDB to store and manage user data, such as profiles or preferences, so that it can be accessed and updated through our Lambda function and API Gateway setup.

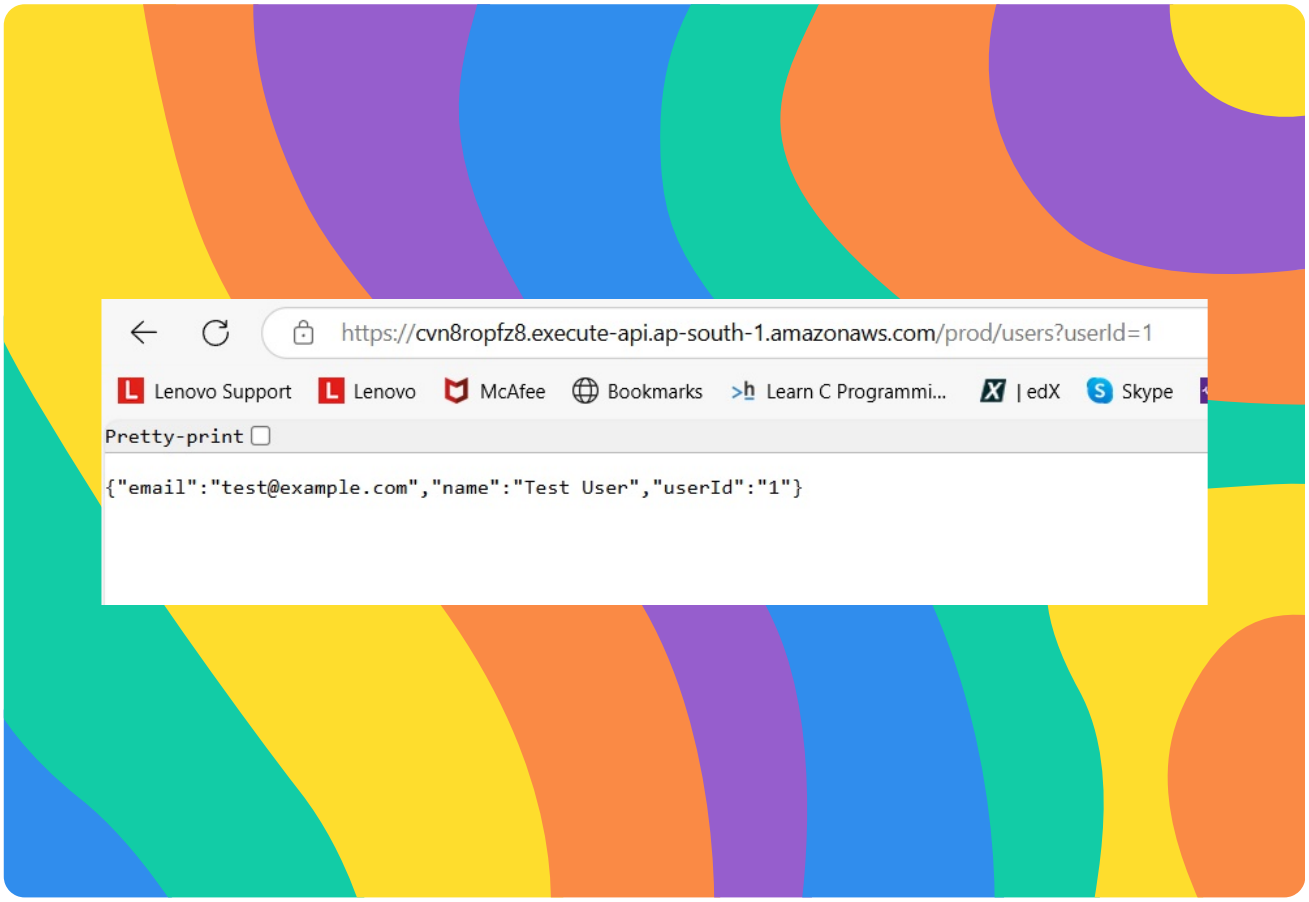

Attributes
View DynamoDB JSON
1 {
2 "userId": "1",
3 "name": "Test User",
4 "email": "test@example.com"
5 }
6

DI
Dineshraj Dhanapathy
NextWork Student

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 my index.html and script.js files to make a request to the API Gateway endpoint because this allows the frontend to communicate with the backend and fetch real-time data from DynamoDB. This step bridges the presentation, logic, and data tiers, enabling dynamic content to be displayed on the website. It ensures that users can interact with the site and receive personalized or updated information seamlessly.

To test my API, I made a GET request to the API Gateway endpoint using the JavaScript fetch() function in my script.js file. The results were successfully displayed on my website, showing the data retrieved from the DynamoDB table. I also tested the endpoint using Postman to verify the response format and status code. This confirmed that the Lambda function and API Gateway were properly integrated and returning the expected data.

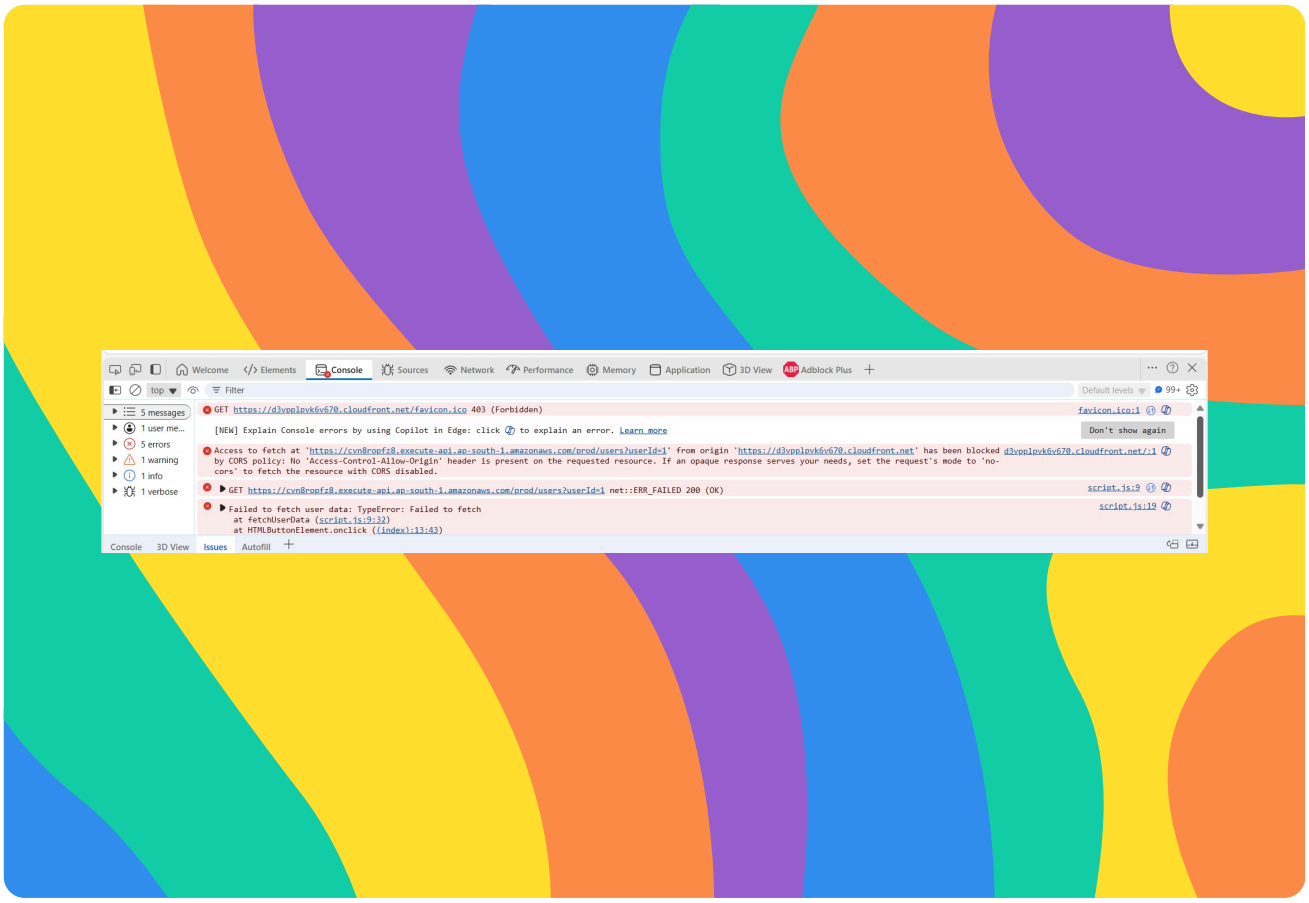

https://cvn8ropfz8.execute-api.ap-south-1.amazonaws.com/prod/users?userId=1
Lenovo Support
Lenovo
McAfee
Bookmarks
Learn C Programmi...
| edX
Skype
Pretty-print
{"email":"test@example.com","name":"Test User","userId":"1"}

Console Errors

The error in my distributed site was because the API URL in line 9 of my script.js file was either incorrect, outdated, or pointing to a non-deployed API Gateway stage. This caused the browser to fail when trying to fetch data from the backend. It's important to ensure that the correct and deployed API Gateway endpoint is used in the script, especially after any changes or redeployments, to avoid broken connections and errors on the frontend.

To resolve the error, I updated script.js by replacing the placeholder [YOUR-PROD-API-URL] with my actual API Gateway Invoke URL. I then reuploaded it into S3 because the website was still referencing the old or incorrect endpoint, causing data fetch failures. Updating the script ensures the frontend makes requests to the correct backend API, allowing the application to retrieve and display user data as expected. To ensure the changes took effect immediately, I created a CloudFront cache invalidation for /script.js, so the latest version would be served to users instead of the cached, outdated one.

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 my CloudFront-hosted frontend. Browsers block requests from different origins by default unless the server explicitly allows it. Since my frontend and backend were on different domains, the browser rejected the API call. To fix this, I needed to enable CORS on my API Gateway so it would allow requests from my CloudFront domain and resolve the error.



DI
Dineshraj Dhanapathy
NextWork Student

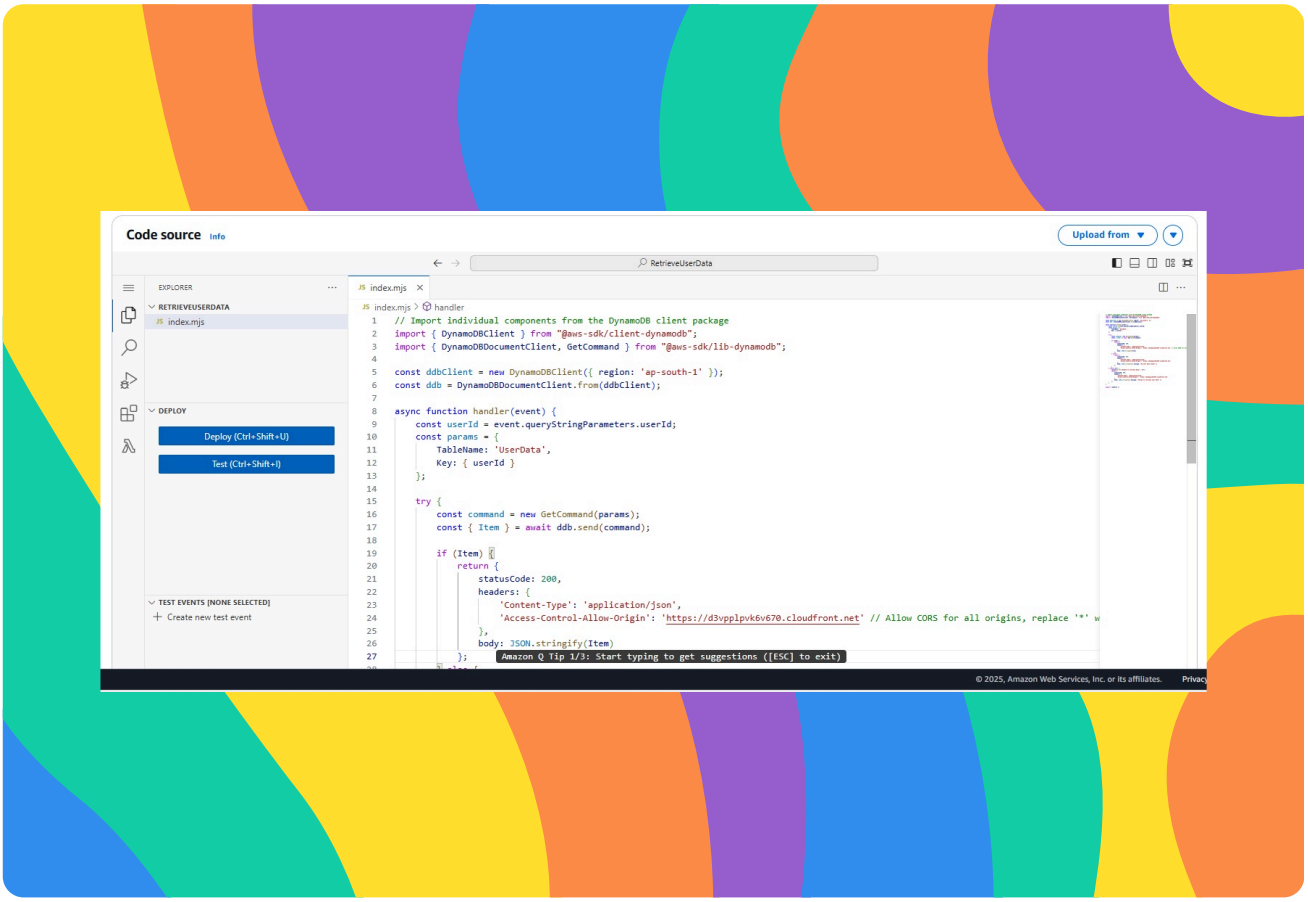
Resolving CORS Errors

To resolve the CORS error, I first navigated to the Resources tab in the Amazon API Gateway console and selected the /users resource. Then, I enabled CORS and set the Access-Control-Allow-Origin value to my CloudFront distribution domain. This update allows my API to accept requests from my frontend hosted on CloudFront. By configuring these CORS settings, I ensured that the browser recognizes my frontend as a trusted source, preventing it from blocking API requests due to cross-origin restrictions.

I also updated my Lambda function because I'm using Lambda Proxy Integration, which requires the Lambda to return the full HTTP response, including the CORS headers. Without these headers in the function's response, the browser would still block the API call, even if CORS is enabled in API Gateway. The changes I made were adding the Access-Control-Allow-Origin header to the response object and replacing * with my CloudFront domain name for better security. This ensures only my frontend can access the API, preventing unauthorized cross-origin requests.

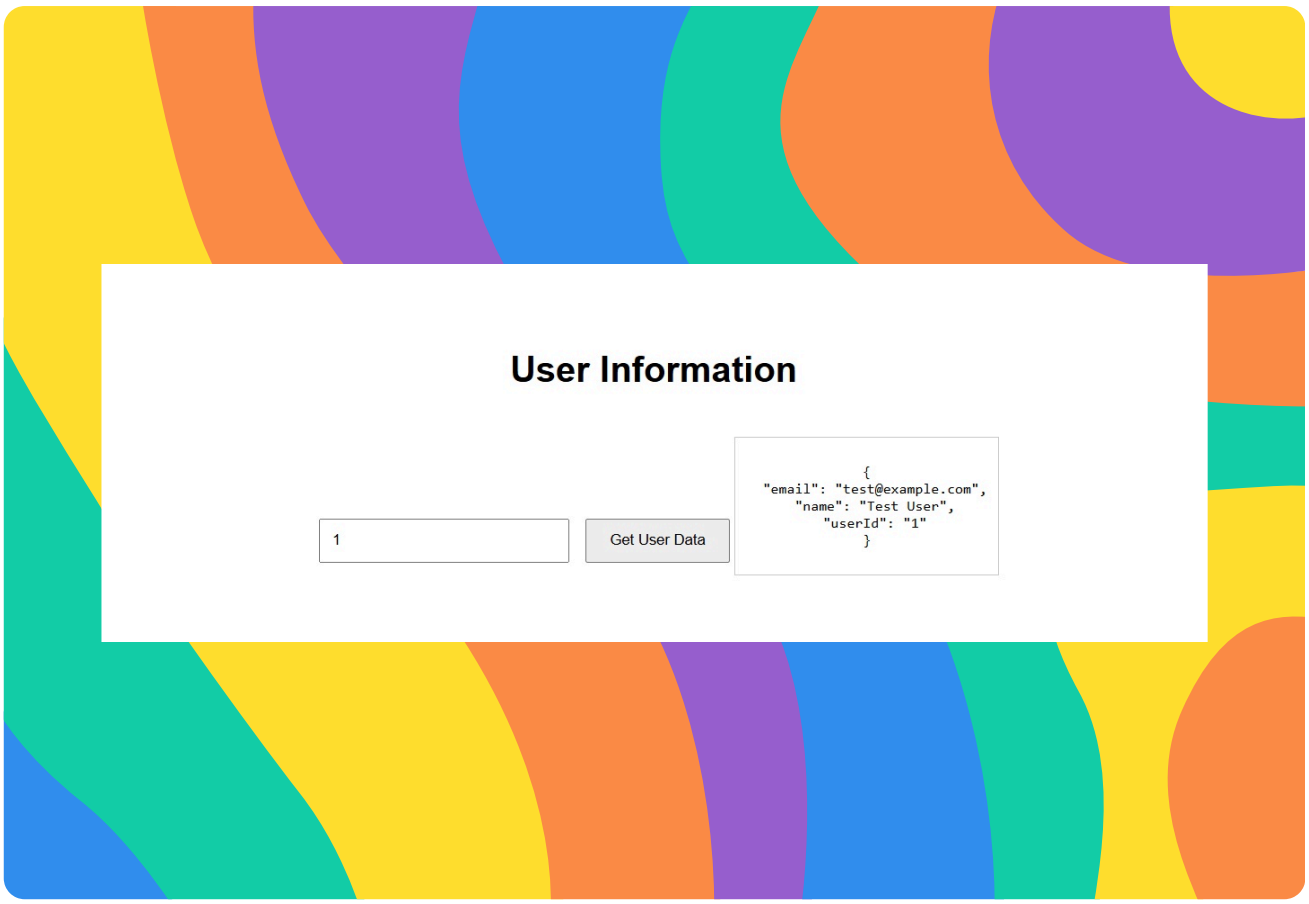
DI

Dineshraj Dhanapathy
NextWork Student



Fixed Solution

I verified the fixed connection between API Gateway and CloudFront by refreshing my CloudFront URL in the browser and performing a test by entering a valid userId and clicking "Get User Data." This time, the request successfully reached the API Gateway, triggered the Lambda function, and returned the expected data from DynamoDB. I also checked the browser's developer console to confirm there were no CORS errors and that the response included the correct headers. Seeing the user data displayed on the site confirmed that all services were properly integrated and working as expected.



NextWork.org

Everyone should be in a job they love.

Check out nextwork.org for more projects