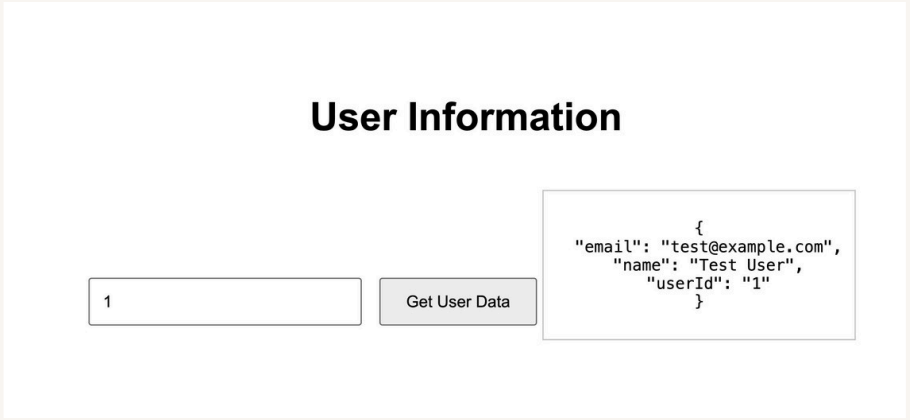
nextwork.org

Build a Three-Tier Web App

Chrispinus Jacob



Chrispinus Jacob
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to set up a three-tier web application from scratch. I will start with the presentation tier, followed by the logic tier, and finally the data tier. I'm doing this project to deepen my understanding of web architecture and how each layer interacts in a full-stack environment.

Tools and concepts

Services I used were Amazon S3, CloudFront, API Gateway, Lambda, and DynamoDB. Key concepts I learnt include Lambda functions, which allow you to run backend code without managing servers; API Gateway, which exposes your APIs to the internet and handles routing and security; and DynamoDB, a serverless NoSQL database used for storing and retrieving user data. I also gained a deeper understanding of CORS, origin access control, and how to connect different AWS services together to build a scalable and secure three-tier web application architecture.

Project reflection

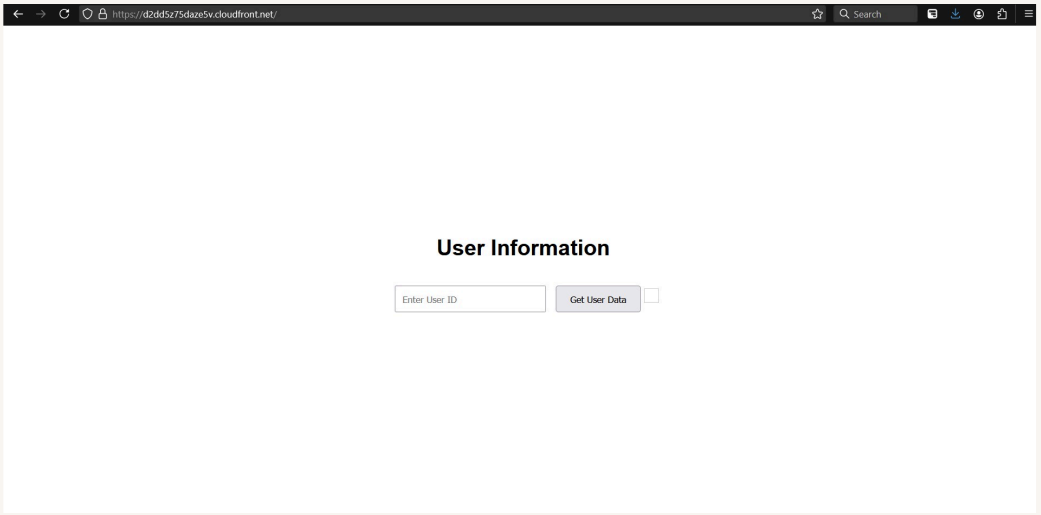
This project took me approximately 2 hours to complete. The most challenging part was troubleshooting the connection between the frontend and the backend—especially dealing with CORS errors and making sure the correct API URL was used in the JavaScript file. It was most rewarding to see the entire flow working smoothly, with data being retrieved from DynamoDB and displayed on the web app through a fully serverless architecture. Seeing all three tiers—presentation, logic, and data—integrated and functioning together was a great learning experience.

I did this project today to gain hands-on experience building a complete serverless web application using AWS services. My goal was to understand how the three-tier architecture works in practice—from hosting static files on S3 and serving them through CloudFront, to handling backend logic with Lambda and managing data with DynamoDB. Yes, this project met my goals because I successfully deployed a functioning web app that connects all three layers, and I now feel more confident in

Presentation tier

For the presentation tier, I will set up how our website will be displayed to our end users because this layer is responsible for delivering the user interface efficiently and securely. I will use an Amazon S3 bucket to host the static website files (HTML, CSS, JavaScript), and configure it with Amazon CloudFront as a content delivery network (CDN). This setup ensures that the website is globally accessible with low latency, high availability, and improved performance through caching and edge locations.

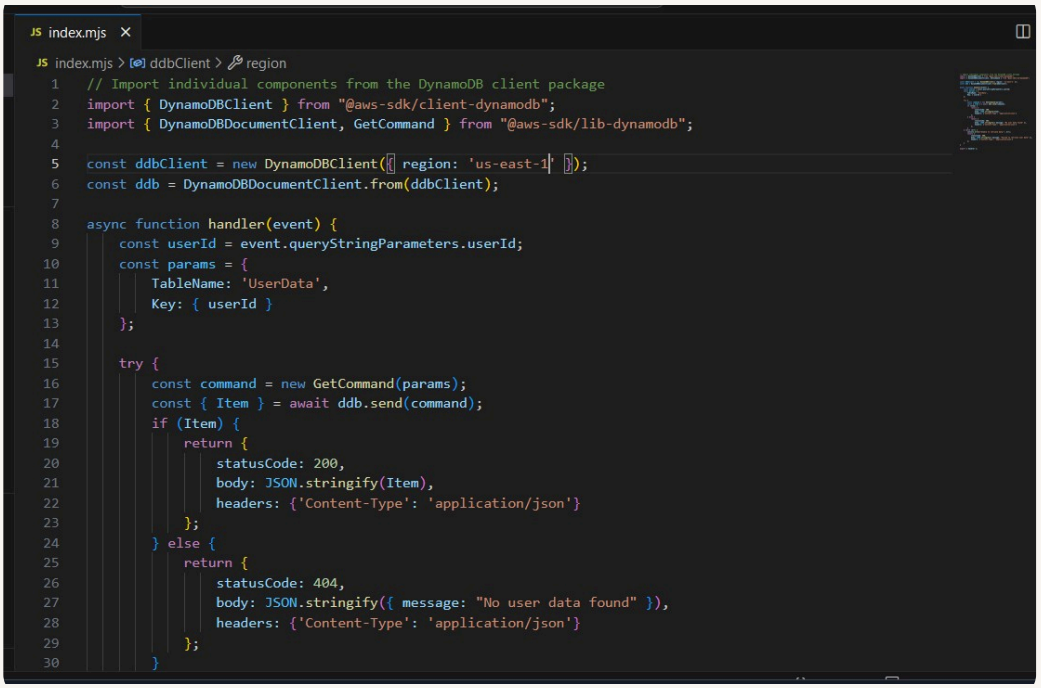
I accessed my delivered website by visiting the CloudFront distribution URL. This distribution URL is working because we also set up Origin Access Control (OAC), which ensures that the S3 bucket only allows access through the CloudFront distribution. This adds a layer of security by preventing direct public access to the S3 bucket and ensures that all traffic is routed through CloudFront, enabling caching, HTTPS support, and better performance for end users.


https://d2dd5z75daze5v.cloudfront.net/
Search
User Information
Enter User ID
Get User Data

Logic tier

For the logic tier, I will set up a Lambda function to process requests such as user search queries. I will also use Amazon API Gateway to receive these requests from the frontend and hand them over to the Lambda function for processing. This is because the logic tier is responsible for handling all dynamic operations and business logic. By using API Gateway and Lambda, we can create a serverless architecture that is scalable, secure, and efficient without the need to manage underlying infrastructure.

The Lambda function retrieves data by looking up a user ID that a user enters over the web app. Inside the function code, we use the AWS SDK, which provides built-in libraries and methods to interact with DynamoDB. By using the SDK, we can easily write queries, use templates, and handle responses efficiently. This allows the function to connect to the database, search for the specific user ID, retrieve the corresponding data, and return it to the frontend through the API Gateway.

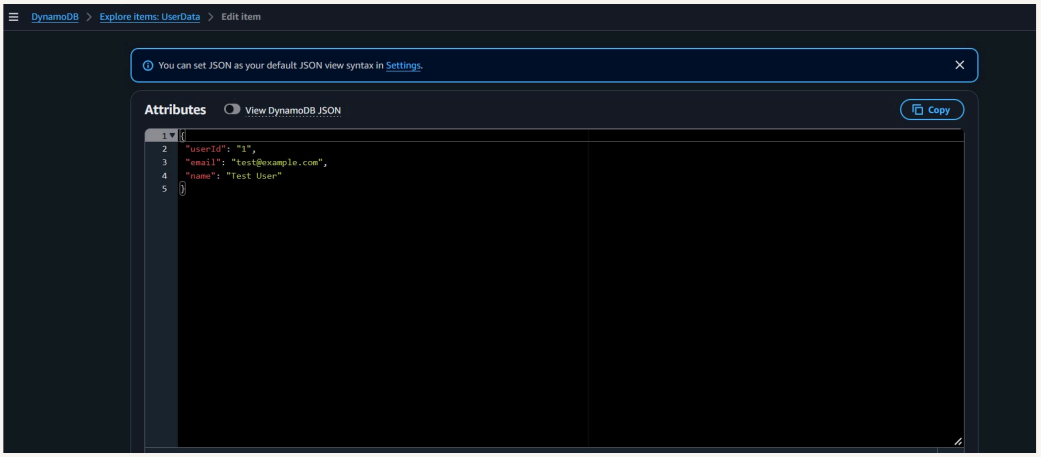

index.mjs
index.mjs > ddbClient > region
// Import individual components from the DynamoDB client package
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";
const ddbClient = new DynamoDBClient({ region: 'us-east-1' });
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'}
};
}

Data tier

For the data tier, I will set up a DynamoDB table that stores user data because, when using our API and looking up user information through Lambda, there is currently no data available to return. DynamoDB will serve as our database to hold user records and other application data. At the moment, since the database has not been configured, there is no user data for the Lambda function to retrieve or return. Once the DynamoDB table is set up and populated, it will enable the logic tier to fetch and respond with actual data to user requests made through the web app.

The partition key for my DynamoDB table is `userId`, which means each item in the table is uniquely identified by a user's ID. This key is used to determine the partition (or physical location) where the data is stored, allowing DynamoDB to quickly locate and retrieve the specific user record when a lookup is made. Using `userId` as the partition key ensures that queries like "get user details by ID" are efficient and scalable, especially when handling large amounts of data.

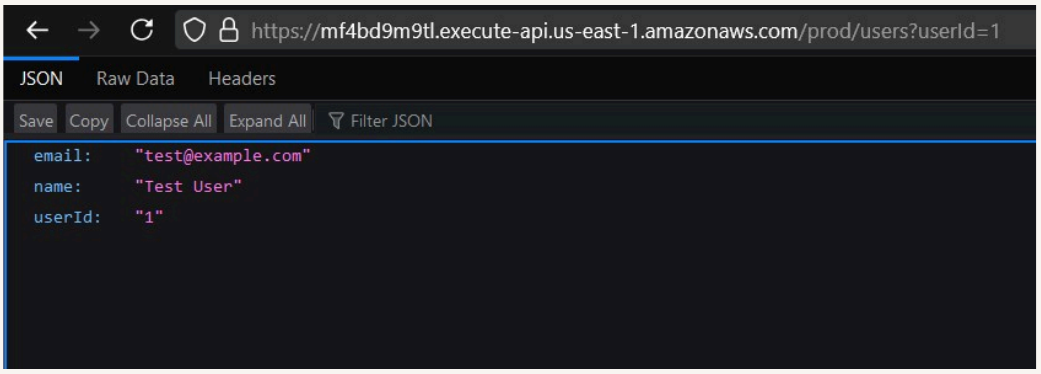
Chrispinus Jacob
NextWork Student



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 because, currently, there is no way for our API to catch the requests that users make through our distributed site. This connection involves integrating the frontend (hosted on S3 and served via CloudFront) with the API Gateway endpoint. By wiring the web app's forms or buttons to send HTTP requests (e.g., using `fetch` or `axios`) to the API Gateway URL, we enable real-time interaction between the user interface and the backend logic handled by Lambda functions.

To test my API, I visited the invoke URL of the prod stage API. This let us test whether we can use the API to retrieve user data. The results were successful—we received some user data in JSON format when we looked up `userId=1`. This proved that the connection between the logic tier (Lambda) and the data tier (DynamoDB) is working correctly, allowing the backend to process the request and return the appropriate data from the database.


https://mf4bd9m9tl.execute-api.us-east-1.amazonaws.com/prod/users?userId=1
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON
email: "test@example.com"
name: "Test User"
userId: "1"

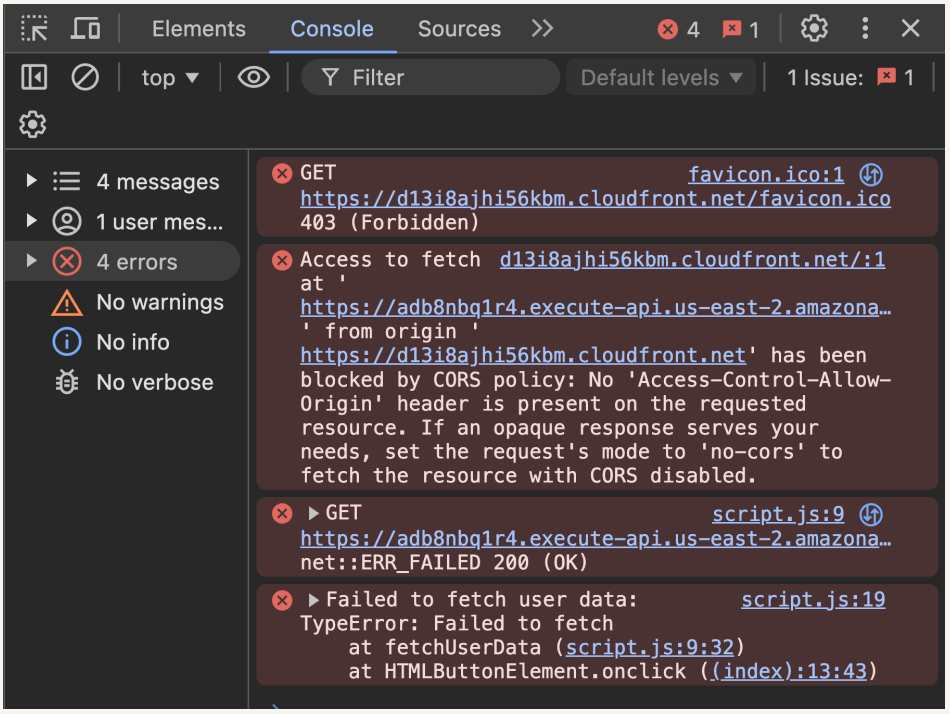
Chrispinus Jacob
NextWork Student

Console Errors

The error in my distributed site was because there was a mistake in the `script.js` file—one of the files I uploaded to S3. This JavaScript file is still referencing a placeholder API URL instead of my actual deployed API Gateway URL. As a result, when users interact with the site, the frontend fails to send requests to the correct backend, leading to no data being returned. Updating the script to point to the correct API endpoint should fix the issue.

To resolve the error, I updated `script.js` by replacing the old placeholder API tag with the actual prod stage invoke URL of the API. I then reuploaded the updated file into the S3 bucket because S3 was still storing and serving the previously uploaded version—the one with the error. Reuploading the corrected version ensures that users accessing the site through CloudFront will now get the updated script that correctly connects to the backend API.

I ran into a second error after updating `script.js`. This was an error on the browser related to CORS (Cross-Origin Resource Sharing) because API Gateway, by default, is only configured to allow requests made directly through its invoke URL in the browser. When the request comes from a different origin—like our website hosted on S3 and served through CloudFront—the browser blocks it unless CORS is properly configured on the API. To fix this, I'll need to enable CORS in the API Gateway settings by allowing the correct origins, methods, and headers in the response.

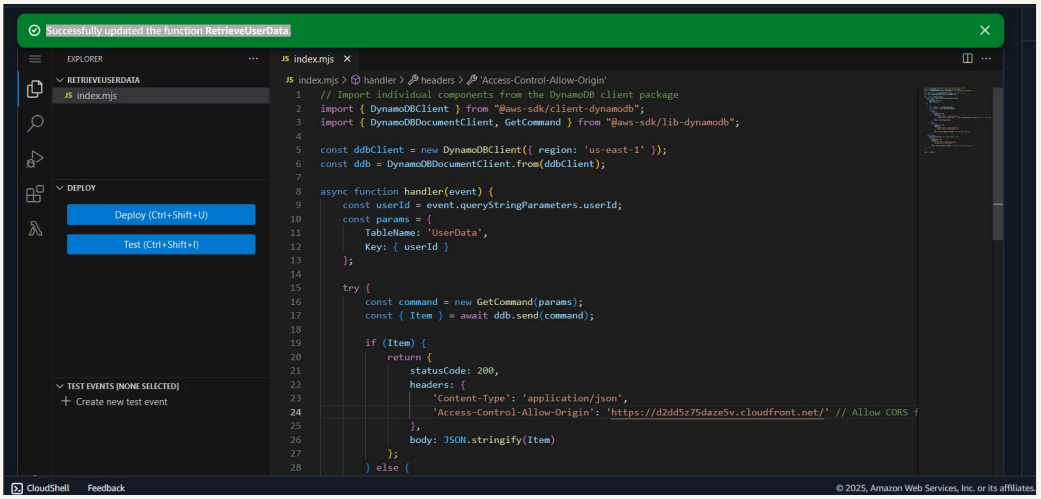

Elements
Console
Sources
4
1
top
Filter
Default levels
1 Issue: 1
4 messages
1 user mes...
4 errors
No warnings
No info
No verbose
GET favicon.ico:1
https://d13i8ajhi56kbm.cloudfront.net/favicon.ico
403 (Forbidden)
Access to fetch d13i8ajhi56kbm.cloudfront.net/:1
at '
https://adb8nbq1r4.execute-api.us-east-2.amazona…
' from origin '
https://d13i8ajhi56kbm.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.
GET script.js:9
https://adb8nbq1r4.execute-api.us-east-2.amazona…
net::ERR_FAILED 200 (OK)
Failed to fetch user data: script.js:19
TypeError: Failed to fetch
at fetchUserData (script.js:9:32)
at HTMLButtonElement.onclick ((index):13:43)


Chrispinus Jacob
NextWork Student

Resolving CORS Errors

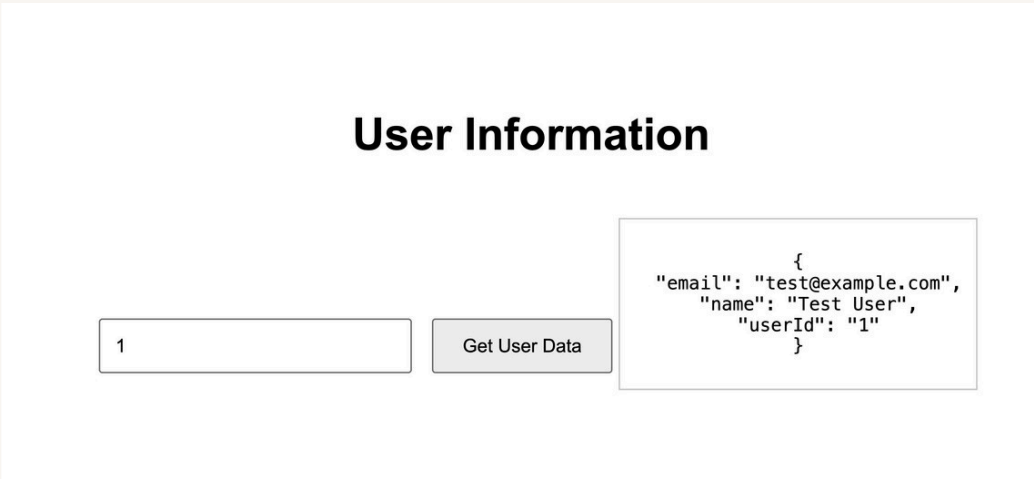
To resolve the CORS error, I first went into our API Gateway and enabled CORS on the `/users` resource. I configured it to allow `GET` requests and made sure to include our CloudFront domain in the list of allowed origins. This ensures that when the browser sends a request from the frontend (hosted on CloudFront), the API responds with the correct CORS headers, allowing the request to go through without being blocked by the browser. This step is essential for enabling communication between the presentation tier and the logic tier when they're served from different domains.

I also updated my Lambda function because it needed to return the appropriate CORS headers in its response. Without these headers, even though CORS was enabled in API Gateway, the browser would still block the response. The changes I made were: * I added `Access-Control-Allow-Origin` to the response headers and set it to my CloudFront domain (or `*` for testing). * I also included `Access-Control-Allow-Headers` and `Access-Control-Allow-Methods` to handle preflight requests if needed. This ensures that the Lambda function works seamlessly with the frontend by allowing the browser to accept responses from a different origin.


Successfully updated the function RetrieveUserData.
EXPLORER
RETRIEVEUSERDATA
index.mjs
DEPLOY
Deploy (Ctrl+Shift+U)
Test (Ctrl+Shift+I)
TEST EVENTS (NONE SELECTED)
Create new test event
index.mjs
index.mjs > handler > headers > 'Access-Control-Allow-Origin'
// 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' });
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,
headers: {
'Content-Type': 'application/json',
'Access-Control-Allow-Origin': 'https://d2dd5z75daze5v.cloudfront.net/' // Allow CORS f
},
body: JSON.stringify(Item)
};
} else {
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.

Fixed Solution

I verified the fixed connection between API Gateway and CloudFront by visiting the CloudFront URL of my site, entering a valid user ID in the input field, and observing that the correct user data was successfully fetched and displayed in the browser. This confirmed that the updated `script.js` file was correctly pointing to the actual API Gateway invoke URL, and that requests from the frontend were being received and processed by the backend. The absence of 403 errors and the presence of real user data in the UI validated that the connection between the presentation and logic tiers was now functioning as expected.



The place to learn & showcase your skills

Check out nextwork.org for more projects