nextwork.org

Build a Three-Tier Web App

Kanika Mathur
github.com/KanikaGenesis

Introducing Today's Project!

In this project, I set out to build a scalable, serverless three-tier web application using AWS. The goal was to gain hands-on experience in creating a modern cloud-native architecture by integrating key AWS services.

This app includes:

- A presentation tier (frontend hosted on S3 and distributed via CloudFront),
- A logic tier (serverless functions using Lambda, triggered by API Gateway),
- And a data tier (user data stored in DynamoDB).

Through this project, I aimed to understand how these services interact and work together to deliver dynamic, responsive application.

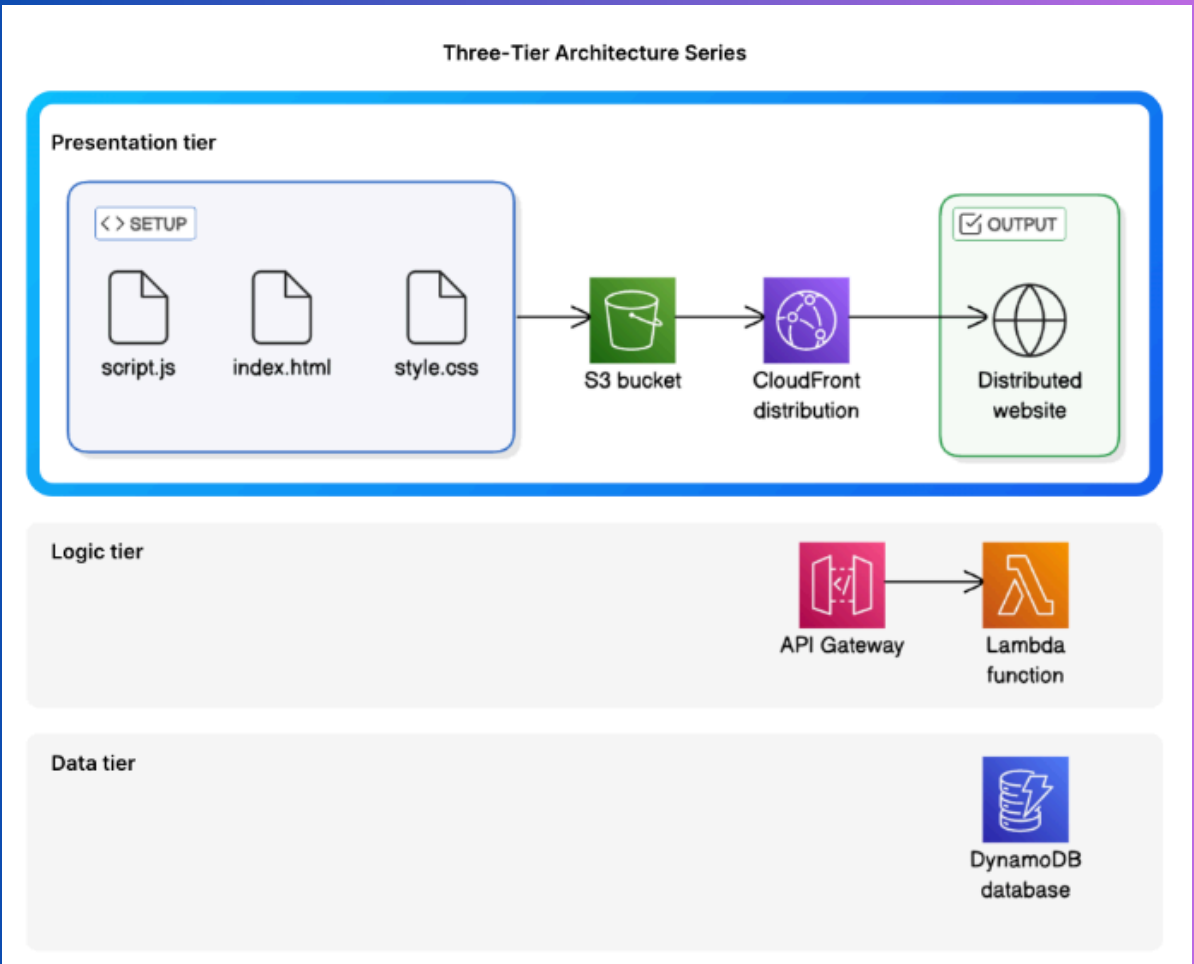
Tools and concepts

AWS Services Used

- Amazon S3 – to host static website files
- CloudFront – to globally distribute content with low latency
- API Gateway – to expose and manage RESTful endpoints
- AWS Lambda – to run backend logic without provisioning servers
- Amazon DynamoDB – to securely store and retrieve user data

Key Concepts Learned

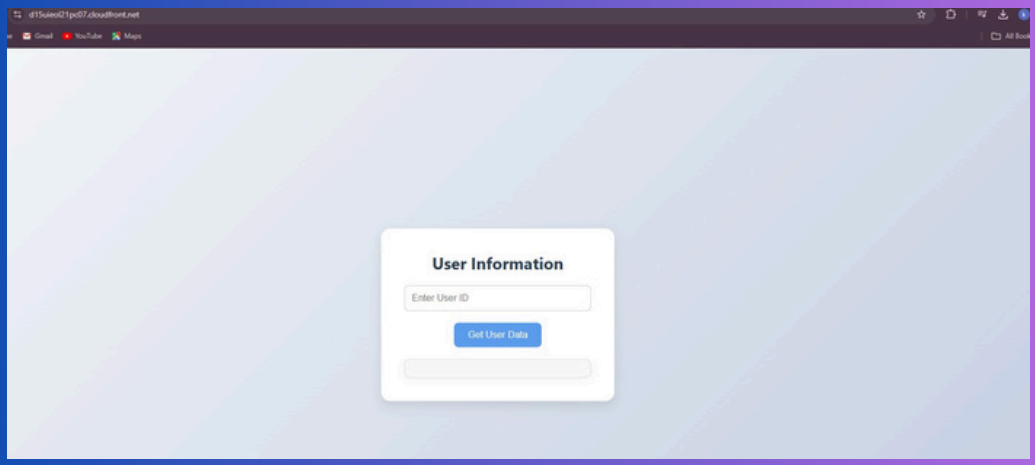
- Serverless architecture with Lambda
- Secure cross-origin requests using CORS
- Three-tier architecture design
- Connecting frontend, logic, and data layers in the cloud


Three-Tier Architecture Series
Presentation tier
SETUP
script.js
index.html
style.css
S3 bucket
CloudFront distribution
OUTPUT
Distributed website
Logic tier
API Gateway
Lambda function
Data tier
DynamoDB database

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.

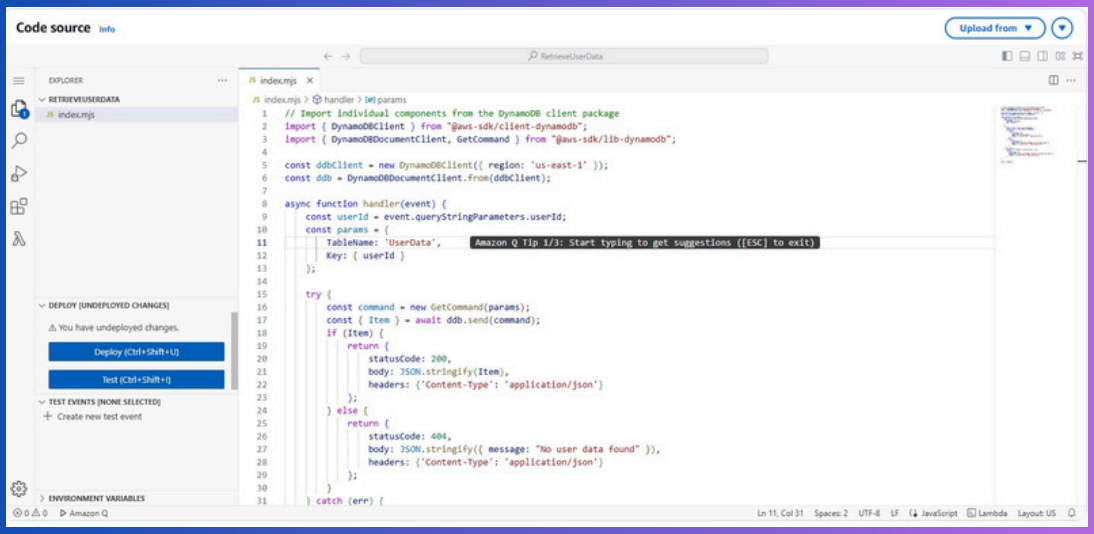
I accessed my delivered website by using the CloudFront distribution URL provided after setup. This URL points to the cached version of my index.html file stored in the S3 bucket. CloudFront ensures that the website loads quickly by delivering content from the nearest edge location, giving users a fast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



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.



Data tier

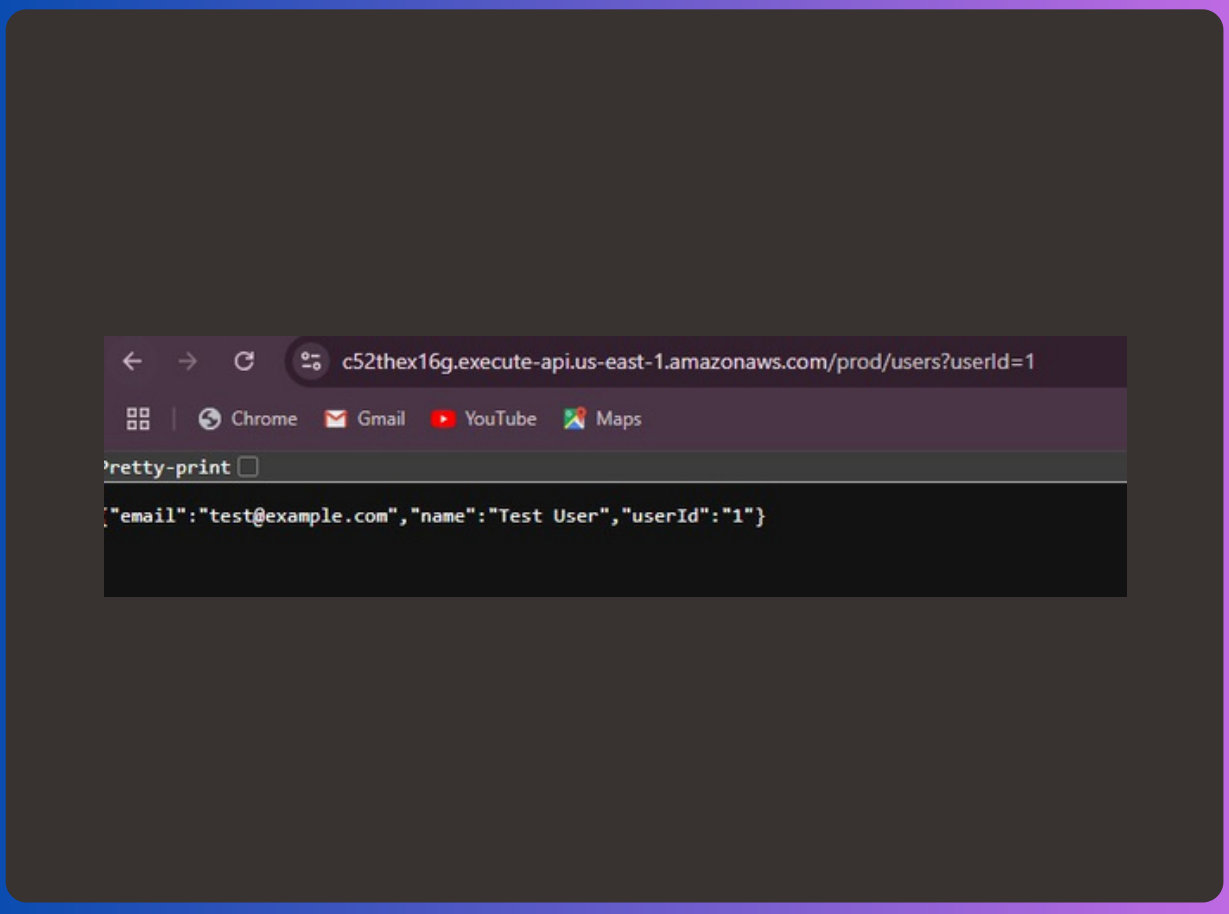
For the data tier, I will set up a DynamoDB table to store and manage user data because it provides a fast, scalable, and serverless NoSQL database solution. By adding user data into the table, my application can retrieve and display information through the Lambda function, completing the backend workflow. 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.

Testing logic and data connection

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 my distributed site!

To test my API, I visited the Invoke URL of the prod stage API. This let us test whether we 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.

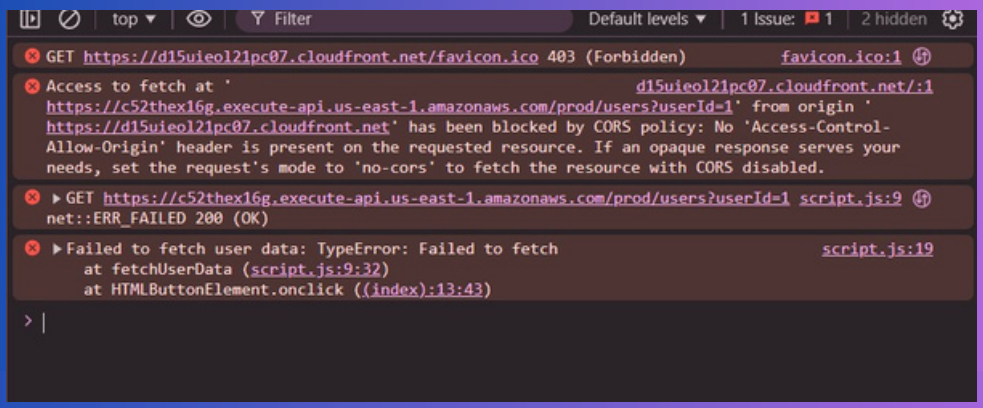


Console Errors

The error in my distributed site was because I hadn't provided the PROD stage API URL in the `script.js` file. This URL is essential for connecting my web application to the API Gateway, allowing it to retrieve user data. Without the correct URL, the application cannot access the backend logic needed to fetch and display the requested information. Updating this URL should resolve the issue and enable the site to function properly.

To resolve the error, I updated script.js by replacing the placeholder with the actual PROD API URL using my code editor on my local machine. I then reuploaded it to S3 because the API Gateway needs the correct URL to respond to user requests. This update will enable the application to access the necessary data and function properly for users.

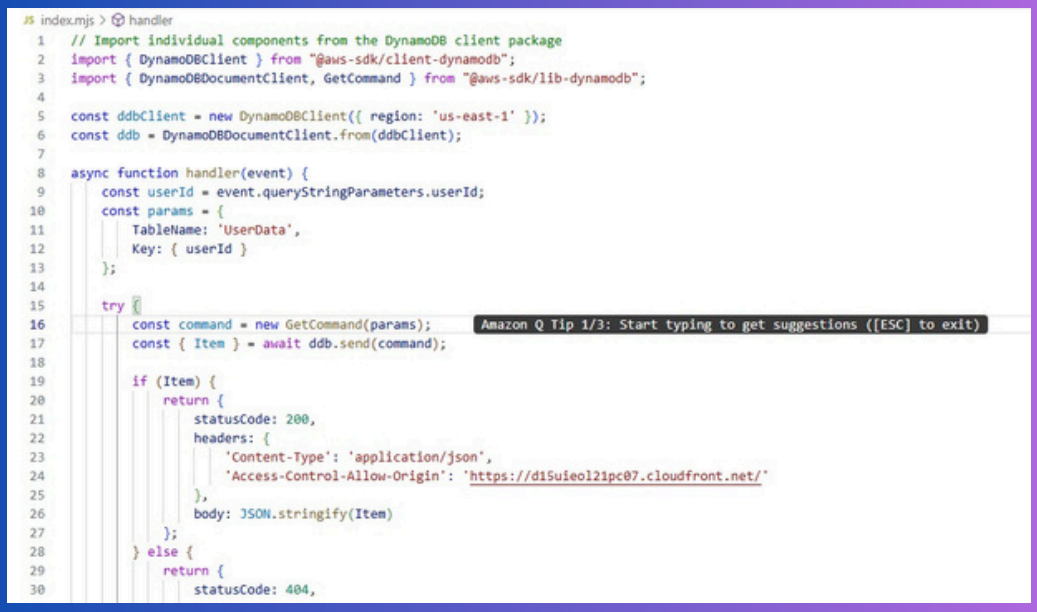
I ran into a second error after updating `script.js`. This was a CORS (Cross-Origin Resource Sharing) error because the API Gateway wasn't configured to allow requests from my CloudFront URL. By default, browsers block requests coming from different domains for security reasons. To fix this, I need to enable CORS on the API Gateway, which will allow my CloudFront-hosted site to communicate with the API and access the necessary data.



Resolving CORS Errors

To resolve the CORS error, I first navigated to the API Gateway console and selected the `/users` resource. I then enabled CORS and checked both GET and OPTIONS under Access-Control-Allow-Methods. Next, I entered my CloudFront distribution domain name as the Access-Control-Allow-Origin value. This configuration allows requests from my CloudFront site to communicate with the API, fixing the CORS issue and enabling proper data access.

I also updated my Lambda function because I needed to ensure that it supports CORS by returning the proper headers in the response. The changes I made included adding the `Access-Control-Allow-Origin` header to allow requests from my CloudFront domain. This ensures that my API can be accessed securely from my frontend, resolving any CORS issues and allowing data retrieval to work correctly. Additionally, confirmed that I replaced the placeholder with my actual AWS region code "us-east-1" for proper functionality.



Final Testing

I verified the fixed connection between API Gateway and CloudFront by refreshing my CloudFront domain in the browser. After making the necessary updates to support CORS and ensuring the Lambda function returned the correct headers, I checked to see if the data from DynamoDB was displayed on my website. The successful retrieval of data confirmed that everything was working properly!

The place to learn & showcase your skills

Check out nextwork.org for more projects